Projective Distribution of XQuery with Updates

Ying Zhang Nan Tang Peter Boncz
CWI, Amsterdam, The Netherlands
{Y.Zhang, N.Tang, P.Boncz}@cwi.nl

***Abstract*—We investigate techniques to automatically decompose *any* XQuery query – including updating queries specified by the XQuery Update Facility (XQUF) – into subqueries, that can be executed near their data sources, i.e., function-shipping. The main challenge addressed here is to ensure that the decomposed queries properly respect XML *node identity* and preserve *structural properties*, when (parts of) XML nodes are sent over the network, effectively copying them.**

We start by precisely characterising the conditions, under which *pass-by-value* parameter passing causes semantic differences between remote execution of an XQuery expression and its local execution. We then formulate a conservative strategy that effectively avoids decomposition in such cases. To broaden the possibilities of query distribution, we extend the pass-by-value semantics to a *pass-by-fragment* semantics, which keeps better track of node identities and structural properties. The pass-by-fragment semantics is subsequently refined to a *pass-by-projection* semantics by means of a novel runtime XML projection technique, which safely eliminates most semantic differences between the local and remote execution of an XQuery expression, and strongly reduces message sizes. Finally, we discuss how these techniques can be used for updating queries, both under the standard W3C XQUF specification, as well as under an extended semantics that allows to update remote documents.

The proposed techniques are implemented in XRPC, a simple yet efficient XQuery extension that enables function-shipping by adding a Remote Procedure Call mechanism to XQuery. Experiments on MonetDB/XQuery establish the performance potential of our XQuery decomposition techniques.

I. Introduction

In this paper, we study ways to decompose any XQUERY query that consults multiple XML documents residing on multiple peers into subqueries that can be executed on those peers, i.e., function shipping. In principle, we do not want to restrict the form of these queries in any significant way: the full W3C recommended XQUERY language [3] including its XQUF extension [7] is the starting point of our decomposition. Our only requirement for peers to participate is running an XML database system (XDBMS) that complies with these W3C recommendations. The goal of this paper is to exploit the computational power of heterogeneous XML engines on the Web to jointly execute XQUERY and XQUF queries.

XQUERY already allows queries over distributed sources through its support for W3C standards, in particular, the ability to open *any* document on the Web through its fn:doc(URI) built-in function. However, the execution model implied by those W3C standards (e.g., HTTP) is data shipping: a full XML document is transported from a remote peer to the querying peer. This means that all query execution happens locally, i.e., at the query originator. It is well known that in many cases this is suboptimal. For instance, an aggregation query on a huge remote XML document that produces only a small result, incurs much less network cost when the aggregation is computed remotely (function shipping) than when the huge XML is shipped to the querying peer (data shipping).

Decomposing queries to address multiple data sources is a well-studied optimisation problem in relational [34], object-oriented [15], [20], and semi-structured databases [30], [31]. While it is natural (and correct) to assume that many of the existing techniques can be carried over, the XML data model and the XQUERY language introduce a number of particular challenges not met elsewhere, that revolve around XML node identities and structural (rather than value-based) relationships between nodes. Previous work on distributed XML [6], [8], [32] only focused on a restricted subset of XQUERY queries, and did not address the problem of transparent query decomposition, such that these challenges did not arise.

Shipping XML Messages. Without loss of generality, we view the subexpressions to be executed by remote peers as XQUERY functions, that may have parameters and produce a result. During *remote* function execution, the calling peer (e.g., query originator) will send a request message containing parameters to a remote peer, which executes the subexpression, and sends back a response message containing the result. To illustrate the challenges of distributing XQUERY, yet preserving XML node identity, consider a subexpression f($a,$b) with two parameters $a and $b of the type node(), that is executed remotely. Complications may arise, for instance, if the subexpression f() tests structural XML relationships among its parameters, such as "$a/parent::b is $b". It therefore depends on the characteristics of the subexpressions f(), and on the way parameters are marshaled in and out of the network messages, to decide whether the distributed query will behave correctly. That is, whether the distributed query is identical to local execution (blindly copying all parameters into the message does not work in this example).

When XML nodes must be shipped over the network, there are two ways to preserve structural relationships. One way is to ship the entire XML document, which defeats the purpose of function shipping. The other way is that pieces/snippets of XML documents must somehow be copied into the messages, changing the "holistic" structural properties and identities of nodes, which may affect the semantics of XQUERY execution on such shipped nodes. Naively, when shipping a node, one would ship its descendants (XML subtree), but other solutions are also possible, and will in fact be proposed in this paper (especially, the idea to use XML projection techniques). In particular, the run-time projection approach contributed in this paper tunes the shape of the shipped XML messages to the characteristics of the query, such that a minimal amount of data is shipped and those structural relationships that are *actually needed* are preserved.

Avoiding Callbacks. One could consider a simple "callback" way of handling XQUERY distribution by not sending XML snippets at all, but just some (global) node identifiers. Each time when a peer needs to execute node-specific XQUERY (XPATH) expressions on such node identifiers, this alternative approach would communicate with the peer where the nodes originally came from, executing the node-specific expressions on that peer and returning the results. While such an approach circumvents semantic problems, it has many drawbacks: *(i)* it basically gives up on the desire to move computation to more powerful peers; *(ii)* it introduces additional network round-trips; *(iii)* it makes all distributed queries – even read-only queries – stateful: a single query might consist of multiple (potentially many) network requests and the query processor on each peer must keep a session context open to guarantee repeatable reads consistency, which causes extra memory consumption and lock contention; and finally *(iv)* there would be additional protocols needed to properly terminate such stateful distributed queries, adding extra protocol complexity, bookkeeping overhead and network latencies. In contrast, the techniques introduced here lead to flexible query distribution where subexpressions can be moved to the peer that can most efficiently process them. Typically, each peer is visited only once, thus network interactions are minimised and peers can handle the subqueries in a stateless manner. Additionally, one could also envision a network protocol that combines "callback" query processing with our techniques, something which might be interesting for handling distributed updating queries (because in the default XQUF semantics only locally stored documents can be updated, hence one would have to "callback" to the peer where a node originated from, to apply the update actions). However, given our target of Internet-wide P2P query processing with high network latencies, we decided against the "callback" approach in our own prototype construction, and fully focused on XQUERY execution by moving XML snippets and computation on them over the network. In this paper we define and solve the problem this approach brings in preserving semantic correctness, and also demonstrate the efficiency of this approach.

XRPC. While our problem statement covers distributed XQUERY in general, this research stems from the particular context of the XRPC project [38], [39]. XRPC adds the concept of Remote Procedure Call to XQUERY by introducing a new statement: execute at {ExprSingle}{Fun(Params)}, where ExprSingle specifies the URI (a constant or a computed one) of the peer, on which Fun will be executed. As XQUERY is a compositional functional language, each query can be chopped up in arbitrary pieces. One can then view the pieces as functions connected together by function parameters and results. With XRPC, we have in principle ultimate flexibility in the way queries can be decomposed, as it allows each function to be executed on an arbitrary peer. An important feature on the network protocol level is *bulk RPC* that allows to handle multiple calls to the same function (with different parameters) in a single network interaction. Bulk RPC is exploited when

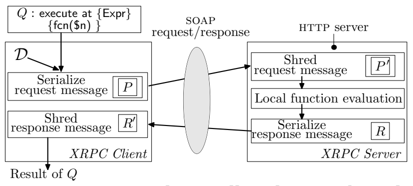


Fig. 1. XQUERY Remote Procedure Call under pass-by-value

a query contains a function call nested in an XQUERY for-loop, which in a naive implementation would lead to as many synchronous RPC network interactions as loop iterations. XRPC is implemented in MonetDB/XQuery [4], an open source XQUERY engine, which we use for experimental evaluation.

XRPC also supports a ***new URI scheme*** xrpc:// in the $uri parameter of the built-in functions fn:put() and fn:doc(). Given a URLxrpc://$\mathcal{P}$/$\mathcal{D}$, fn:put() stores the XML tree rooted at its $node parameter on the remote peer $\mathcal{P}$ as document $\mathcal{D}$, which possibly overwrites the existing $\mathcal{D}$. With fn:doc(), $\mathcal{D}$ could then be retrieved (over HTTP) from (the XRPC server on) peer $\mathcal{P}$. As we will see in Section VII, this extension enables supporting updates on remote documents identified by xrpc:// URIs.

Figure 1 shows a query Q that performs a single XRPC function call to fcn(), with a single parameter (a node $n from some document $\mathcal{D}$). To make an XRPC call, the local peer formulates a SOAP request message, which contains a deep copy P of the node $n. The Simple Object Access Protocol (SOAP) is an XML-based message format commonly used by web services [22]. XRPC follows the previously mentioned approach of copying the XML subtree of a node parameter, which implies a *pass-by-value* parameter passing strategy. The message is sent as a synchronous HTTP POST request. The remote peer runs an HTTP server, which parses the request message and constructs a separate XML fragment for each node parameter (in this example a single fragment P'). The remote peer then evaluates the function and serialises the result into a response message (here, a deep copy of the result node, denoted R). Finally, the local peer parses the response message and constructs a separate XML fragment for each node-typed result (here R'), which is the result of Q.

Problem Statement. Our goal is to rewrite an XQUERY Q that uses XML documents with xrpc:// URIs stored at remote peers, into an equivalent query Q' that uses XRPC calls to execute parts of the query (expressed as XQUERY functions) on those remote peers. For a query Q, $Q(\mathcal{D})$ denotes the result of evaluating Q over a (possibly distributed) database $\mathcal{D}$. Two queries Q and Q' are *equivalent*, if $Q(\mathcal{D}) = Q'(\mathcal{D})$ for any given database $\mathcal{D}$ (under the XQUERY *deep-equal* semantics).

We illustrate XQUERY decomposition as follows:

```
for $e in doc("employees.xml")//emp
where $e/@dept = doc("xrpc://example.org/depts.xml")//dept/@name
return $e
```

the URL xrpc://example.org/depts.xml implies that the remote peer example.org supports XRPC, so the predicates could be pushed as:

```
declare function fcn($n as xs:string) as xs:boolean
{ $n = doc("depts.xml")//dept/@name };
for $e in doc("employees.xml")//emp
where execute at { "example.org" } { fcn($e/@dept) } return $e
```

In this example, the parameter and return value of the function fcn() are of atomic types. In more complex cases, nodes may be involved, such that potential semantic differences due to pass-by-value should be considered (discussed in Section II), which is our main challenge.

Contributions & Road map. Section II identifies the semantic differences of remote XQUERY pass-by-value function evaluation with respect to standard, local function evaluation. Section III describes an XQUERY CORE based query decomposition framework. This leads in Section IV to a conservative XQUERY decomposition strategy that avoids semantic problems simply by refraining from decomposition in all problem cases. To make our rewrites more effective and robust against syntactic variation, we describe normalisation and code motion rewrite strategies. As a second contribution, Section V extends the pass-by-value semantics with a new *pass-by-fragment* message format that conserves more structural relationships between nodes passed in a message and allows more predicates to be distributed. Section VI introduces a new *runtime XML projection* technique, which we use to generate messages that conserve all needed structural relationships between transferred XML nodes, and thus allow even more freedom in query decomposition. As a runtime technique, it is able to prune XML data much more than previously described compile-time projections [2], [5], [18]. In Section VII-A, we discuss how updating queries can be handled, both in the normal XQUF semantics, as well as under an extension, in which we allow non-local documents to be updated. An evaluation of the performance benefits of our techniques is given in Section VIII. We discuss related work in Section IX and conclude in Section X with outlook on future work.

```
declare function makenodes() as node()
{⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩/b };                 ▷ node ⟨b⟩⟨c/⟩⟨/b⟩ has parent::a

declare function overlap($l as node(), $r as node()) as boolean
{ not(empty($l//∗ intersect $r//∗)) };            ▷ are $l and $r related?

declare function earlier($l as node(), $r as node()) as node()
{ if ($l≪$r) then $l else $r };

let $bc := makenodes(),
    $abc:=$bc/parent::a                            ▷ $bc has a parent $abc
return  (for $node in ($bc, $abc)
          let $first := earlier($bc, $abc)                 ▷ always $abc
          where overlap($first, $node)                  ▷ always overlap
          return $node)//c                          ▷ returns only one ⟨c/⟩
```

TABLE I
EXAMPLE QUERY Q_1

II. SEMANTIC DIFFERENCES OF PASS-BY-VALUE

There are well-defined semantic differences [38] between evaluating an XQUERY expression locally and executing it remotely under pass-by-value parameter passing. We discuss these differences with a query Q_1 in Table I. This query evaluates three functions: makenodes(), overlap() and earlier().

Problem 1: Non-downward XPath Steps. Reverse and horizontal XPATH axis navigation (e.g., parent, ancestor, preceding(-sibling) and following(-sibling)) from remote function parameters always produces empty results, as pass-by-value node serialisation only includes the descendants of a node inside the message. Consider the following:

```
let $bc := execute at {“example.org”} {makenodes()},
    $abc := $bc/parent::a
```

here, $abc evaluates to the empty sequence, instead of the correct a-node $\langle a\rangle\langle b\rangle\langle c/\rangle\langle /b\rangle\langle /a\rangle$.

It is possible to evaluate downward XPATH steps on a sequence of remote nodes, but only if we are sure that these nodes are ordered *and* non-overlapping (otherwise, the results of such XPATH steps will fail to respect node identity and order, as described below).

Problem 2: Node Identity Comparisons. If a remote function returns a sequence with two identical nodes, or two identical nodes are passed as function parameters, pass-by-value represents them as two different copies. This leads to the duplicate elimination problem described just above, and any *node identity* comparison will always yield false. For instance:

```
where execute at {“example.org”}{overlap($first, $node)}
```

yields false, while the local query evaluation gives true.

Problem 3: Document Order. The parameters of a function call on a remote peer are serialised into the message in parameter order, in separate XML fragments. Even if the parameter nodes are disjoint (making *Problem 2* irrelevant), the relative order between these XML fragments may differ from their original order. Thus, inter-parameter node comparisons (“≪”, “≫”) may behave differently from the local semantics. Consider the usage of earlier() in Q_1 as:

```
let $first := execute at{“example.org”} {earlier($bc,$abc)}
```

In both iterations, the variable $first binds to a copy of $bc, instead of $abc, although $abc is the parent of $bc.

Another problem with document order, not revealed by this example, could occur when comparisons of nodes from different XML documents are executed on remote peers. The XQuery/XPath Data Model (XDM) [12] defines that the relative order of nodes in different documents is *implementation-dependent*, but must be *stable* during the processing of the same query. Consider the query

```
declare function earlier2() as boolean
{ doc(“xrpc://a.example.org/a.xml”)/a ≪
            doc(“xrpc://b.example.org/b.xml”)/b };
execute at {“a.example.org”}{earlier2()} =
      execute at {“b.example.org”}{earlier2()}
```

which, depending on how documents are ordered by the remote peers, could return true or false[1], while XDM requires it to always return true. Note, however, that a query containing a *single* call to earlier2() may return either true or false, confirming XDM, then in such queries, earlier2() could be executed at a remote peer.

Problem 4: Interaction Between Different Calls. Additional semantic differences can occur when XQUERY subexpressions (sequences) may contain nodes that were obtained as results from *different* remote function calls, and these function calls, directly or indirectly, accessed the same XML document on some peer. Node sequences can become intermixed by any XQUERY construct that accepts multiple inputs, namely: sequence construction, and the built-in functions union, except, and intersect. A special source of call-mixing is the return clause of a for-loop in which remote function evaluation is performed, because the return clause implicitly creates a sequence that concatenates the expression result of all loop iterations (each of which performed a semantically separate remote function call). The result of such “mixed-call expressions” is

[1]Even if the two calls to earlier2() were executed on the same remote peer, without any guarantees for consistency, the results could be different, since each call is a separate query on the remote peer.

that nodes returned by different calls may in fact stem from the same document. However, node identity and ordering between nodes from different calls is not preserved, leading to semantic differences. For example, even if a downward XPATH step is applied on an input sequence containing nodes obtained from different remote calls, the result can have the wrong order (placing the results from the first call always before those of the second call) and will fail to properly eliminate duplicates:

```
(for $node in ($bc, $abc)
let $first := execute at {“example.org”}{earlier($node,$abc)}
return $node)//c
```

The above two XRPC calls produce nodes belonging to separate XML fragments. Under pass-by-value, evaluating //c produces two separate copies of c nodes, while in local execution the nodes returned from earlier() are from the same XML fragment, such that XPATH steps return a duplicate-free result.

Problem 5: XQuery Built-in Functions. Various problems may occur when evaluating certain built-in functions remotely.

1) static-base-uri(), default-collation() and current-datetime(): depend on the *static* XQUERY context.
2) base-uri() and document-uri(): depend on the *dynamic* context of node expressions.
3) root(): accesses the document root.
4) id() and idref(): return all nodes in a document with certain ID/IDREF values.
5) lang(): accesses the xml:lang attribute of the context node and its ancestors.

Class 1 of above built-in functions is handled by extending the XRPC message format with extra attributes such that the remote side can declare identical values for these context attributes[2]. Class 2 is dealt with by adding these properties as attributes in the XRPC nodes (such as xrpc:element) that enclose serialised parameter/result nodes in the SOAP messages. Use of the fn:base-uri() and fn:document-uri() in XRPC is substituted by xrpc:base-uri() and xrpc:document-uri() wrappers that take these attributes into account when invoked on XRPC parameter nodes. As solutions for Class 1-2 are available, the main problem with built-in functions is posed by Classes 3-5, which access non-descendants of parameter nodes, and thus cannot be supported with pass-by-value.

In the remainder, we present decomposition techniques and extensions to enhance the pass-by-value semantics, that solve the aforementioned problems.

III. XQUERY CORE REWRITE FRAMEWORK

XQUERY CORE [10] (abbreviated XCORE) is a subset of XQUERY, in which all implicit operations are made explicit. We adopt a subset of XCORE expressions in Table II, which is sufficient to capture XPATH 1.0 and XQUERY FLWOR expressions [10]. Additionally, we support all updating expressions (rule $UpdExpr$) and the transform expression (TransformExpr) as defined by XQUF. For reasons of space, we thus omit repeating grammar rules for those expressions. We use a representation of XPATH paths in our XCORE grammar that keeps consecutive steps together, rather than nesting each step

```
1:  Expr        ::= “()” | ExprSingle | ExprSeq
2:  ExprSeq     ::= “(” ExprSingle (“,” ExprSingle)∗ “)”
3:  ExprSingle  ::= Literal | VarRef | ForExpr | LetExpr | IfExpr | CompExpr | StepExpr | UpdExpr
                  | OrderExpr | NodeSetExpr | Constructor | FunCall | Typeswitch | TransformExpr
4:  VarRef      ::= “$”Var
5:  Var         ::= “$”QName
6:  ForExpr     ::= “for” Var “in” Expr “return” Expr
7:  LetExpr     ::= “let” Var “:=” Expr “return” Expr
8:  IfExpr      ::= “if” “(” Expr “)” ThenElse
9:  ThenElse    ::= “then” Expr “else” Expr
10: Typeswitch  ::= “typeswitch” “(”Expr“)” CaseClause+ “default” Var “return” Expr
11: CaseClause  ::= “case” Var “as” SequenceType “return” Expr
12: CompExpr    ::= Expr (ValueComp | NodeCmp) Expr
13: ValueComp   ::= “=” | “! =” | “<” | “<=” | “>” | “>=”
14: NodeCmp     ::= “is” | “≪” | “≫”
15: OrderExpr   ::= Expr “order by” OrderSpecs
16: OrderSpecs  ::= Expr (“ascending” | “descending”)(, OrderSpecs) ∗
17: NodeSetExpr ::= Expr NodeSetOp Expr
18: NodeSetOp   ::= “union” | “intersect” | “except”
19: Constructor ::= (“document” | “text”) “{” Expr “}”
                  | (“element”|“attribute”) (QName | “{”Expr“}”) “{”Expr“}”
20: StepExpr    ::= “/” | AxisStep “::” NodeTest
21: AxisStep    ::= RevAxis | FwdAxis | HorAxis
22: RevAxis     ::= “ancestor” | “ancestor-or-self” | “parent”
23: FwdAxis     ::= “self” | “child” | “attribute” | “descendant” | “descendant-or-self”
24: HorAxis     ::= “preceding” | “preceding-sibling” | “following” | “following-sibling”
25: NodeTest    ::= “node()” | “text()” | QName | “∗”
26: FunCall     ::= QName “(” (Expr (“,” Expr) ∗)? “)”
27: UpdExpr     ::= InsertExpr | DelteExpr | RenameExpr | ReplaceExpr
```

TABLE II
XCORE GRAMMAR RULES

basic XQUERY query	

(let \$s := doc(“xrpc://A/students.xml”)/people/person,
      \$c := doc(“xrpc://B/course42.xml”),
      \$t := \$s[tutor = \$s/name]
 for \$e in \$c/enroll/exam where \$e/@id = \$t/id return \$e)/grade
	Q_2

XCORE variant	

(let \$s := doc(“xrpc://A/students.xml”)/child::people/child::person return
   let \$c := doc(“xrpc://B/course42.xml”) return
      let \$t := for \$x in \$s return
         if (\$x/child::tutor = \$s/child::name) then \$x else ()
      return for \$e in \$c/child::enroll/child::exam return
                  if (\$e/attribute::id = \$t/child::id) then \$e else ())/child::grade
	Q_2^c

normalised XCORE variant	

(let \$t := (let \$s := doc(“xrpc://A/students.xml”)/child::people/child::person
                  return for \$x in \$s return
                               if (\$x/child::tutor = \$s/child::name) then \$x else ())
 return for \$e in (let \$c := doc(“xrpc://B/course42.xml”)
                        return \$c/child::enroll/child::exam)
              return if (\$e/attribute::id = \$t/child::id) then \$e else ())/child::grade
	Q_2^n

TABLE III
EXAMPLE QUERY Q_2

in a separate for-loop (when allowed – the use of position() precludes this). Such an optimisation is common in XQUERY engines, and is part of XQUERY normalisation, further described in Section IV. Additionally, we define two new rules for the XRPC extension [38]:

```
28: XRPCExpr  ::= “excute at” “{”ExprSingle“}” “function” XRPCParam “{”Expr“}”
29: XRPCParam ::= “()” | “(” “$”Var “:=” VarRef (“,” XRPCParam)? “)”
```

Rule 28 identifies an xrpc:// URI in expression ExprSingle, and declares a new *anonymous function* that is to be executed remotely. It is noticeable that these grammar rules lack the expressive power to define recursive functions. This does not matter for XQUERY decomposition, as our decomposition strategies will not generate recursive functions. We also note that the syntax defined by rules 28 and 29 differs from the actual XRPC syntax (execute at {ExprSingle}{FunApp(ParamList)}). The syntax used here is only for presentation purpose, to avoid the need to define all rules concerning declaration of user-defined functions. Thus, our simple XCORE rule without explicit user-defined function declarations allows to express all queries in a single ExprSingle, which in turn can be mapped to a query graph. This simplifies the formulation of analysis steps.

A. XCore Dependency Graph

We introduce a dependency graph (d-graph) for an XCORE query. Consider the XQUERY query Q_2 in Table III, which asks for the grade in course42 of students having a tutor who is also a student, and its XCORE equivalence Q_2^c.

[2]If static-base-uri() is not set, we ship the value xrpc://$\mathcal{P}$/doc/, so that fn:doc() calls with a relative document URI call back to the originating peer $\mathcal{P}$.

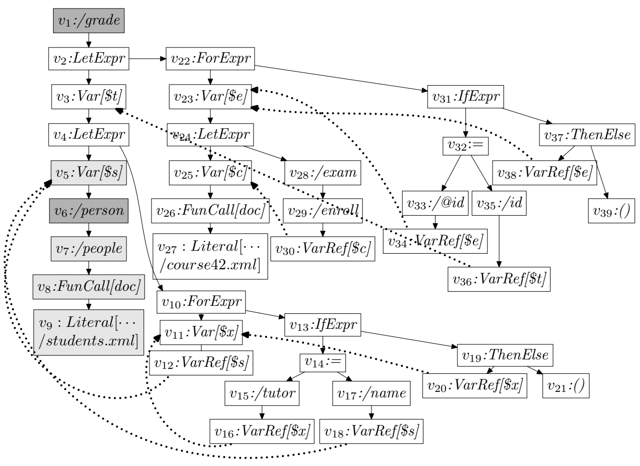


Fig. 2. *d*-graph of the normalised XCORE variant Q_2^n in Table III

A *dependency graph* is a directed, ordered and connected graph G with vertices $V(G)$ and edges $E(G)$. Each vertex v is denoted as v_i*:rule[val]*, where v_i is a unique vertex identifier, *rule* is the grammar rule represented by v_i, and *val* is an optional value indicating the right-hand-side of *rule*. There is a single root vertex without incoming edges. $E(G)$ consists of parse edges $E_p(G)$ and varref edges $E_v(G)$. Each *parse edge* is an ordered vertex pair (u, v), where u corresponds to a parsing rule r_u that directly causes the use of another parsing rule r_v. A *varref edge* is an ordered vertex pair (w, x) denoting a variable usage. When a VarRef rule is used, an additional edge is created between the VarRef vertex and the Var vertex that defines the variable.

Example 3.1: *Figure 2 shows the* d*-graph of Q_2^n in Table III. Solid and dashed lines represent parse and varref edges, respectively. The variable binding in the first* let *expression corresponds to vertices $v_2, \dots, v_{21}$, and vertices $v_{22}, \cdots, v_{39}$ depict its* return *clause. The edge (v_6, v_7) is a parse edge. The edge (v_{30}, v_{25}) is a varref edge, as the variable used by v_{30} is a reference of variable $\$c$ introduced by v_{25}. Thus, a d-graph is in essence a parse-tree with additional (dashed) edges to indicate variable usages.*

We define three types of dependency relationships upon the reachability between two vertices x, y in $V(G)$: (1) x *"parse-depends on"* y, denoted as $x \overset{p}{\rightsquigarrow} y$, if y is reachable from x via *only* parse edges; (2) x *"varref-depends on"* y, denoted as $x \overset{v}{\rightsquigarrow} y$, if y is reachable from x via at least one varref edge; and (3) x *"depends on"* y, denoted as $x \rightsquigarrow y$, if either $x \overset{p}{\rightsquigarrow} y$ or $x \overset{v}{\rightsquigarrow} y$ holds. The compositional nature of XQUERY means that $x \rightsquigarrow y$ concisely captures all semantic dependencies between subexpressions. Consider Figure 2, $v_{15} \overset{p}{\rightsquigarrow} v_{16}$, since (v_{15}, v_{16}) is a parse edge; $v_{15} \overset{v}{\rightsquigarrow} v_{11}$, as v_{11} is reachable from v_{15} via $(v_{15}, v_{16}), (v_{16}, v_{11})$ and (v_{16}, v_{11}) is a varref edge.

For a *d*-graph G and a vertex $r_s \in V(G)$, we use the term *subgraph* to mean the vertex-induced subgraph of r_s, denoted G_{r_s}, including r_s and all $u \in V(G)$ where $r_s \overset{p}{\rightsquigarrow} u$; r_s is called the *root* of the subgraph. For instance, the subgraph rooted at vertex v_{22} contains vertices $v_{22}, \cdots v_{39}$, but does not contain vertices $v_3, \dots, v_{21}$. Throughout this paper, we use the terms (sub)graph and (sub)query interchangeably, as a (sub)query is represented by the induced subgraph rooted at some vertex.

B. XRPCExpr Insertion

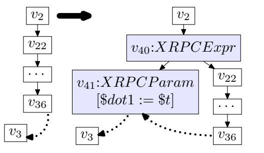


Fig. 3. XRPCExpr Insertion

We can decide to evaluate a certain subgraph G_{r_s} rooted at r_s remotely over XRPC, by inserting a v_x:XRPCExpr node above it. This may only be done if we can ensure that the result of the rewritten query is identical to the original query. Such an insertion means that a new function will be defined that contains G_{r_s} as its body. In the main query graph, G_{r_s} is replaced by a remote XRPC call to this function, which receives as parameters all variable references in G_{r_s} that resolve to variable bindings outside G_{r_s}:

1) Insert a vertex v_x:XRPCExpr, a parse edge (v_x, r_s), and replace each incoming edge (v_{in}, r_s) with a new edge $(v_{in}, v_x)^3$.
2) For each outgoing varref edge from vertex $v_i \in V(G_{r_s})$ to $v_j \in V(G) \backslash V(G_{r_s})$, where edge $(v_i, v_j) \in E_v(G)$ is a varref edge as (v_i:VarRef[$\$qname$], v_j:Var[$\$qname$]), we insert a new vertex v_k, a new parse edge (v_x, v_k) and replace the varref edge (v_i, v_j) by (v_i, v_k) and (v_k, v_j). Here, v_k has the form v_k:XRPCParam[$\$p$:=$\$qname$], which introduces a new variable $\$p$ and binds it to $\$qname$ in v_j.
3) If there are no outgoing edges as stated in *step* 2, we insert a vertex v_l with the form v_l:XRPCParam[()] (i.e., empty parameter), and a parse edge (v_x, v_l).

Example 3.2: *Consider the* d*-graph in Figure 2. Suppose that the subgraph rooted at v_{22} is identified for an* XRPCExpr *insertion (Figure 3). First, insert vertex v_{40} and replace edge (v_2, v_{22}) by (v_2, v_{40}) and (v_{40}, v_{22}). For the outgoing varref edge (v_{36}, v_3), vertex v_{41} is inserted below v_{40} and the varref edge is replaced by two new varref edges: (v_{36}, v_{41}), (v_{41}, v_3).*

IV. Conservative Decomposition

In this and the next two sections, we first describe algorithms to decompose read-only XCORE queries. We delay the discussion of decomposing queries containing any UpdExpr and TransformExpr expressions until Section VII.

By-value Insertion Conditions. Given a *d*-graph G and a subgraph G_{r_s} of G rooted at r_s, under the pass-by-value semantics, vertex r_s is in the set $I(G)$ of *valid decomposition points* (*d*-points), iff r_s satisfies all of the following conditions:

i. $\nexists n \in V(G): n.rule \in \{$RevAxis, HorAxis$\} \wedge ($useResult$(n, r_s) \vee$ useParam$(r_s, n))$;

ii. $\nexists n \in V(G): n.rule \in \{$NodeCmp, NodeSetExpr$\} \wedge ((r_s.rule \in \{$NodeCmp, NodeSetExpr$\} \wedge n \notin V(G_{r_s})) \vee$useResult$(n, r_s) \vee$ useParam$(r_s, n))$;

iii. $\nexists n \in V(G), \exists m \in V(G) : n.rule =$ AxisStep $\wedge$ $m.rule \in \{$ForExpr, OrderExpr, ExprSeq, NodeSetExpr, AxisStep$\backslash\{$parent, preceding-sibling, following-sibling, self, child, attribute$\}\}$ $\wedge(($useResult$(n, m) \wedge m \rightsquigarrow r_s) \vee$ (useResult$(n, r_s) \wedge m \in V(G_{r_s})) \vee (m \notin V(G_{r_s}) \wedge r_s \overset{p}{\rightsquigarrow} n \overset{v}{\rightsquigarrow} m))$;

iv. $\nexists n \in V(G) : n.rule =$ FunCall $\wedge$ $n.val \in \{$fn:root(), fn:id(), fn:idref(), fn:lang()$\} \wedge ($useResult$(n, r_s) \vee$ useParam$(r_s, n))$.

[3]Determined by the algorithms (Section IV-VI) that compute the insertion points (i.e., determine if a vertex may be an r_s), r_s is the only vertex in the subgraph G_{r_s} that has incoming edges from vertex outside G_{r_s}.

where we impose these restrictions symmetrically both on expressions that use the result of the remote expression r_s, and on the way remote expressions (below r_s) use their shipped parameters:

$$\text{useResult}(n, r_s) \Leftrightarrow n \rightsquigarrow r_s$$
$$\text{useParam}(r_s, n) \Leftrightarrow n \in V(G_{r_s}), \exists v \in V(G) \backslash V(G_{r_s}) : n \rightsquigarrow v$$

Conditions i and ii guard against using any node comparisons, and horizontal and reverse XPATH steps on shipped nodes, avoiding *Problems 1-3* (Section II). Condition ii also disallows decomposing any node comparisons, when a query contains multiple such expressions, to avoid the problem with document order of nodes from different documents. Condition iii avoids using XPATH steps on shipped nodes stemming from expressions that might be so-called "mixed-call sequences" (ForExpr, ExprSeq, NodeSetExpr), avoiding *Problem 4*. It also guards against sequences not in node order (ForExpr, OrderExpr) or with nodes that may be overlapping (the restrictions on NodeSetExpr and XPATH steps). This ensures that downwards XPATH steps can be used on shipped node sequences that are ordered and non-overlapping. Condition iv states that shipped nodes may not be used as parameters of the listed built-in functions (*Problem 5*).

Example 4.1: *In the* d*-graph of example query Q_2^n (Figure 2), we mark in grey colours the* d*-points identified by the conservative decomposition strategy. The* XPATH *step* /grade *that is performed on the result of a* for*-loop, matches condition iii and causes all vertices that depend on v_{10} and v_{22} (the* ForExpr*s) as well as all their descendants to be excluded from $I(G)$, leaving v_1 and the subgraphs rooted at v_5 as* d*-points.*

Interesting Decomposition Points. While a *d*-point may be semantically valid, remote evaluation of the subquery below it might be senseless. Consider the *d*-point v_8, which contains only an fn:doc() function call in its subgraph. Executing this function remotely provides no performance gain, as it only demands the shipping of a whole document. Similarly, remote execution of expressions that do not involve any XML documents should be avoided. Therefore, we filter *d*-points by first annotating each vertex $v_x \in V(G)$ with the URI *dependency set* $D(v_x)$. Here, $D(v_x)$ represents the set of URIs that are used as parameters of fn:doc() in vertices that the vertex v_x can reach via parse edges:

$$D(v_x) = \{uri :: v_y | \{v_y, v_z\} \in E(G) : uri = v_z.val \wedge v_x \overset{p}{\rightsquigarrow} v_y \wedge$$
$$v_y.rule = \text{FunApp} \wedge v_y.val = \text{“doc”} \wedge v_z.rule = \text{Literal}\}$$

Note that we tag each uri with the vertex v_y where the document is opened, to be able to distinguish the use of the same document through multiple fn:doc() calls. This definition does not cover the case that the parameter of fn:doc() is an expression instead of a literal. In those cases, we use a wildcard symbol "$*$" as uri. In this paper, the built-in function fn:collection() is treated as an fn:doc($*$), and an element construction is assigned an artificial unique URI fn:doc(v_i::v_i).

One can use the URI dependency set to partition the $V(G)$ into *equivalence classes*, i.e., those vertices with the same URI dependency set belong to the same class. Using all vertices in an equivalence class, we can consider its induced subgraph in G, and try to handle it in a single XRPC subquery. Thus, we define *interesting decomposition points* (*i*-points) $I'(G)$ as those valid insertion points that *(a)* are a root vertex in their induced subgraph (if the root node happens to be a Var vertex, we consider its value expression instead as root), *(b)* contain at least one fn:doc() and *(c)* execute at least one XPATH step on the fn:doc() function:

$$I'(G) = \{v_x | v_x \in I(G) : \nexists v_y : v_y \overset{p}{\rightsquigarrow} v_x \wedge D(v_x) = D(v_y) \wedge$$
$$\exists v_z : v_x \overset{p}{\rightsquigarrow} v_z \wedge v_z.rule = \text{AxisStep} \wedge \exists \text{xrpc://}uri \in D(v_x)\}$$

This definition is also used by the next two algorithms to filter *d*-points.

Example 4.2: *In Figure 2, the two subtrees rooted at v_5 and v_{25} correspond to two different equivalent classes $D(v_5) = \{$*xrpc://A/students.xml :: *$v_9\}$ and $D(v_{25}) = \{$*xrpc://B/course42.xml :: *$v_{27}\}$. However, v_{25} is not a valid insertion point. The vertices in $I'(G)$ (coloured dark grey) are v_6 (the highest non* Var *vertex in the subtree rooted at v_5) and the root v_1. Thus, $I'(G) = \{v_1, v_6\}$.*

Normalisation. Rewriting algorithms that operate on the XCORE level are vulnerable to syntactic variation. In the case of our decomposition strategy, an important vulnerability comes from the behaviour of the strategy to ship subgraphs consisting of parse-edges only. That is, varref-edges are not pushed, but rather become parameters to the function. The syntactic freedom one has in XQUERY of defining subexpressions, e.g. inline or via a variable reference to a previous let-binding, therefore affects our strategy. For this purpose, as part of XCORE normalisation, we re-order let-bindings, moving them as deep into the query as possible. More specifically, let-bindings are moved to just above the lowest common ancestor vertex (defined in terms of parse-edges) of all vertices that reference its variable. The query Q_2^c (Table III) can be normalised to Q_2^n (Table III), which can thus be rewritten as Q_2^v in Table IV.

The main achievement of normalisation in the above case is to relate the call to doc("../course42.xml") through parse-edges (directly calling $\$c$ in Q_2^n), instead of varref edges (referencing $\$c$ in Q_2^c), with its use in the /child::enroll/child::exam XPATH steps. However, these being part of a ForExpr with the /grade step on top, causes insertion condition ii to prohibit pushing it. In the next section on pass-by-fragment, however, we will see that normalisation was not in vain, and the query can be decomposed into Q_2^f (Table IV).

Distributed Code Motion. The let-normalisation phase has the effect of pushing expressions that depend on the same documents downwards, potentially below an interesting insertion point (which makes them be executed remotely). However, it can happen that some of the expressions initially found below an interesting insertion point can in fact better be moved above it (to be executed locally). In particular, it is safe to assume that expressions that solely depend on a parameter of a function, can better be evaluated on the caller side. Moving a subexpression out of a function can be done by passing that subexpression as an additional parameter to the function. With pass-by-value passing, such a rewrite may not always be safe, however if only *d*-points are moved, the technique is semantically safe. Analogous to well-known compiler technique of moving invariant statements out of the loop (and its use in parallel processing [17]) we call this technique *distributed code motion*.

Q_2^v: decomposed Q_2^n under pass-by-value

```
declare function fcn1() as node()*
{doc("xrpc://A/students.xml")/child::people/child::person};

declare function fcn0() as node()*
{(let $t := let $s := execute at{'A'}{fcn1()} return
              for $x in $s return if ($x/child::tutor = $s/child::name) then $x else ()
  return for $e in (let $c := doc("xrpc://B/course42.xml")
                      return $c/child::enroll/child::exam)
        return if ($e/attribute::id = $t/child::id) then $e else () )/child::grade};

execute at {} {fcn0()}
```

Q_2^f: decomposed Q_2^n under pass-by-fragment

```
declare function fcn1() as node()*
{let $s := doc("xrpc://A/students.xml")/child::people/child::person return
  for $x in $s returnif ($x/child::tutor = $s/child::name) then $x else () };

declare function fcn2($para1 as node()) as node()*
{for $e in (let $c := doc("xrpc://B/course42.xml")
             return $c/child::enroll/child::exam)
  return if ($e/attribute::id = $para1/child::id) then $e else () };

declare function fcn0() as node()*
{let $t := execute at {'A'}{fcn1()}
  return (execute at {'B'}{fcn2($t)}) / child::grade};

execute at {} {fcn0()}
```

Applying Distributed Code Motion in Q_2^f

```
declare function fcn2new($para2 as xs::string∗) as node()*
{for $e in (let . . . return . . . ) return if ($e/attribute::id = $para2) then $e else ()};

declare function fcn0() as node()*
{let $t := execute at {'A'}{fcn1()} return
    let $l := $t return (execute at {'B'}{fcn2new($l/child::id)})/child::grade};
```

TABLE IV
QUERY DECOMPOSITION AND CODE MOTION

Example 4.3: *Consider the function* fcn2() *in Table IV, we may observe that the expression* $para1/child::id *only depends on the function parameter* $para1*. Shipping full* person *nodes* $para1 *from peer* A *to* B*, only to extract the string value of its* id *child at* B*, may waste bandwidth, especially if* person *carries much more data than just an* id*. Instead, it would be better to extract the string value of* id *at peer A and only ship the strings. This optimisation can be realised by adding a new parameter* $para2 *to the function, and substituting* $para1/child::id *in the body with it. In the function* fcn0() *that calls* fcn2new()*, we save the original function parameter* $t *in a new* let*-binding* $l*, and pass* $l *instead of* $t*. The additional function parameter is passed as* $l/child::id*. Finally, the affected function parameter* $para1 *is no longer used, so we remove it, arriving at the result as the code motion part in Table IV.*

V. BY-FRAGMENT DECOMPOSITION

The node copying done by pass-by-value is the main source of semantic differences. This, in turn, leads to serious restrictions in the way the decomposition strategy can push expressions remotely. For this reason, we extend the pass-by-value message passing semantics into a new *pass-by-fragment* message passing semantics that better preserves structural relationships of XML nodes. The basic idea is to avoid serialising the same nodes twice, by grouping all node-valued data in the message in a preamble element fragments. In principle, each node parameter is serialised below a separate fragment child element. However, if a sent node is a descendant of another one, it is not serialised twice, as we can reuse the XML fragment of the other node. We also ensure that the XML fragments are sorted in original document order, which means that ancestor/descendant relationships in the same message, as well as node identity and document order, are preserved.

Later in the message, where XQUERY sequences are serialised (inside sequence tags), we just provide references to

Excerpt of a request message with pass-by-value

```
⟨call⟩
  ⟨sequence⟩⟨element⟩⟨b⟩⟨c⟩⟨/b⟩⟨/element⟩⟨/sequence⟩
  ⟨sequence⟩⟨element⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/element⟩⟨/sequence⟩
⟨/call⟩
```

Excerpt of pass-by-fragment message for earlier($bc,$abc)

```
⟨env:Envelope ...⟩
  ⟨env:Body⟩
   ⟨request⟩
    ⟨fragments⟩ ⟨fragment⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/fragment⟩ ⟨/fragments⟩
    ⟨call⟩
     ⟨sequence⟩⟨element fragid="1" nodeid="2"⟩⟨/sequence⟩
     ⟨sequence⟩⟨element fragid="1" nodeid="1"⟩⟨/sequence⟩
    ⟨/call⟩
   ⟨/request⟩
  ⟨/env:Body⟩
⟨/env:Envelope⟩
```

Fig. 4. By-value vs. By-fragment Messages

the nodes that were previously serialised in the fragments. In particular, element tags, which are used to contain as a child the fully serialised copy of a node, now just carry two numeric attributes, fragid and nodeid. In order to keep XRPC an interoperable protocol that is easy to implement for XQUERY engines and the XRPC WRAPPER [38], node referencing is also expressible in XQUERY. Supposing $msg is the root of the message, with $fragid and $nodeid numbers, we can identify the referenced nodes as follows:[4]

$msg//fragment[$fragid]/descendant::node()[$nodeid]

Example 5.1: *Going back to* Q_1 *in Table I, the lower part of Figure 4 shows the* XRPC *request message sent for the call* execute at {"example.org"} {earlier ($bc, $abc)} *from the discussion of Problem* 3*. Recall that the node* $bc *with value* $\langle b\rangle\langle c/\rangle\langle /b\rangle$ *is contained in the* $abc *fragment* $\langle a\rangle\langle b\rangle\langle c/\rangle\langle /b\rangle\langle /a\rangle$*. The lower part of the figure shows an excerpt of the message as produced for pass-by-fragment. Here, both node parameters* $bc *and* $abc *are represented in* element *nodes with* fragid *and* nodeid *attributes. The* XQUERY *engine handling the call will use these attributes to evaluate:*

$bc := $msg:fragment[1]/descendant::node()[2],
$abc := $msg:fragment[1]/descendant::node()[1]

such that earlier($bc,$abc) *correctly returns* $abc*, because* $abc $\ll$ $bc*, just like on the peer that invoked this function. The upper part, with the changed part of the old pass-by-value message (element* call*), shows that node parameters were previously repeatedly serialised, causing node order and identity relationships between parameters to be lost.*

We made a conscious choice not to rely on ID/IDREF for referencing nodes, since this would require adding ID attributes to the XML data in the fragments. As XRPC is designed to respect and conserve XML SCHEMA type information, this would cause the XRPC message to no longer respect user-defined schemas.

By-fragment Insertion Conditions. Given a d-graph G and a subgraph G_{r_s} of G rooted at r_s, under the pass-by-fragment semantics, vertex r_s is in the set $I(G)$ of valid decomposition points, iff r_s satisfies all of the following conditions:

I. $\nexists n \in V(G) : n.rule \in \{\text{RevAxis}, \text{HorAxis}\} \wedge (\text{useResult}(n, r_s) \vee \text{useParam}(r_s, n))$;

II. $\nexists n \in V(G) : n.rule \in \{\text{NodeCmp}, \text{NodeSetExpr}\} \wedge ((r_s.rule \in \{\text{NodeCmp}, \text{NodeSetExpr}\} \wedge n \notin$

[4]Note that descendant::node() does not return attribute nodes. We use the nodeid of its parent and include the name of the attribute in an attribute element, so it can be found back with an additional attribute step.

ProjectionPath	::=	**doc** "(" Literal "::" Literal ")" ("/" *SimplePath*)*
SimplePath	::=	*Axis NodeTest* | *SimplePath* "/" *Axis NodeTest*
Axis	::=	"self::" | "child::" | "attribute::"
		| "descendant::" | "descendant-or-self::"
		| "**ancestor**::" | "**ancestor-or-self**::" | "**parent**::"
		| "**root()**" | "**id()**" | "**idref()**" | "**lang()**"
NodeTest	::=	((*NCName*|∗) :)?(*NCName*|∗)| "node()" | "text()"

TABLE V
GRAMMAR RULE EXTENSION OF *ProjectionPath* (BOLD)

$V(G_{r_s})$ ∧ hasMatchingDoc(n, r_s)) ∨ ((useResult(n, r_s) ∨ useParam(r_s, n)) ∧ hasMatchingDoc(n, n)));

III. $\nexists n \in V(G), \exists m \in V(G) : n.rule =$ AxisStep ∧ $m.rule \in$ {ForExpr, ExprSeq, NodeSetExpr} ∧ ((useResult(n, m) ∧ $m \rightsquigarrow r_s$) ∨ (useResult(n, r_s) ∧ $m \in V(G_{r_s})$) ∨ $(m \in V(G) \backslash V(G_{r_s}) \wedge r_s \overset{p}{\rightsquigarrow} n \overset{v}{\rightsquigarrow} m)$) ∧ hasMatchingDoc$(m, m)$;

IV. $\nexists n \in V(G) : n.rule =$ FunCall ∧ $n.val \in$ {fn:root(), fn:id(), fn:idref(), fn:lang()} ∧ (useResult(n, r_s) ∨ useParam(r_s, n)).

Thus, with the pass-by-fragment semantics, we modify the pass-by-value decomposition conditions listed in Section IV by restricting the prohibitions to decompose a node r_s formulated in Conditions ii and iii to only those r_s, for which the predicate hasMatchingDoc() holds:

$$\text{hasMatchingDoc}(v_1, v_2) \Leftrightarrow \forall uri_l{::}v_i \in D(v_1), \exists uri_r{::}v_j \in D(v_2) : v_i \neq v_j \wedge (uri_l = uri_r \vee uri_l = * \vee uri_r = *)$$

By stating that the given expressions depend on two *different* applications of fn:doc() with the *same* URI (taking into account computed URIs as wildcards), this predicate precisely isolates the problem of creating result sequences with remote nodes from multiple calls to the same document.

The ForExpr is a special form of combining the results of multiple calls. A remote call nested in a for-loop, which depends on the same remote document, is treated as a single call, since Bulk RPC ensures that all iterations of the remote call nested in the for-loop, are handled in a single message exchange (where pass-by-fragment now ensures proper conservation of node relationships). Finally, we remove from condition iii the restrictions that arbitrary ordering (OrderExpr) cannot be used and that all pushed AxisSteps should be of the non-overlapping kind (parent, preceding-sibling, following-sibling, self, child, and attribute), as the pass-by-fragment message passing is able to properly conserve sequence order and the ancestor/descendant relationships between transported nodes. As the remaining problems with mixed-call sequences are related to dealing with multiple network message exchanges in the same query, this problem can not be solved inside the message passing semantics alone and is beyond our current scope. The restrictions to avoid horizontal and reverse XPATH steps on remote nodes (Condition I) and on using built-in functions (Condition iv) will be addressed in the next section.

Example 5.2: *Consider Figure 2, as the constraint* hasMatchingDoc() *in condition III does not hold, all vertices in the graph are identified as valid decomposition points under the pass-by-fragment semantics. However, most vertices will be filtered out by the definition of interesting decomposition points, which leads to* $I'(G) = \{v_1, v_2, v_4, v_6, v_{22}, v_{24}\}$.

VI. BY-PROJECTION DECOMPOSITION

The basic idea of using XML projection [21] is, for a given XQUERY query Q and an XML document $\mathcal{D}$, to extract a minimal part of data $\mathcal{D}'$ needed to execute Q such that $Q(\mathcal{D}) = Q(\mathcal{D}')$. The projection technique conducts a compile-time path analysis on Q, to derive a set of simple path expressions that over-estimate the nodes that Q touches. These simple paths are referred to as *projection paths*. Here, a *projection path* is an XML path that starts from the document root, containing forward navigation but not predicates (e.g., $doc(\$uri)/a/b/@id$). Projection paths consist of returned paths and used paths. *Returned paths* describe the nodes that are returned by the expression. *Used paths* indicate the nodes necessary to answer the query but are never returned as results (e.g., predicates). Based on the projected paths $\mathcal{P}$ of query Q from path analysis, a loading algorithm is applied to $\mathcal{P}$ and an XML document (from a file or a stream) $\mathcal{D}$. A projected XML document (or stream) $\mathcal{D}'$ is then generated, which contains all used and returned nodes plus the descendants of the returned nodes, and is queried with Q.

There are three reasons why projecting XML is extremely interesting for distributed XML processing: *(i)* until now, when sending nodes, we had to serialise all descendants – which potentially contain huge subtrees that may remain untouched on the other side. This amounts to wasted network bandwidth as well as serialisation and shredding effort. *(ii)* if documents are projected into lean skeletons that only contain the relevant portions, it becomes feasible to serialise XML fragments from some *lowest common ancestor* on, possibly even the document root. Even with pass-by-fragment, the execution of reverse/horizontal XPATH axes on remote nodes is impossible. By extending projecting XML with support for reverse and horizontal axes, however, we get a tool to precisely identify the lowest common ancestor of an XML document that needs to be included to allow correct remote execution of those axes. *(iii)* the projection technique can even be applied to support the built-in functions fn:root(), fn:id(), fn:idref() and fn:lang(), i.e., by taking the lowest common ancestor of those, if a path contains one of these functions. For these reasons, we further refine the pass-by-fragment message passing semantics into a so-called *pass-by-projection* semantics. XML projection can be used in both directions: to project the parameters in a request message, and to project the function's result sequence before shipping back the response.

Insertion Conditions. Pass-by-projection removes the by-fragment insertion conditions (in Section V) I and IV, such that only II and III, i.e., the application of node comparison, node set operators and axis steps on top of multiple calls to fn:doc() with the same URI, remains illegal.

Message Extension: Projection Paths. We introduce an optional element as a sub-element of a request tag: projection-paths, which in turn has zero or more child elements returned-path and used-path. In the new pass-by-projection semantics, the absence or presence of this element determines whether the response message should be in the original pass-by-value or the new pass-by-projection format.

Example 6.1: *To illustrate projected* XRPC *messages, the*

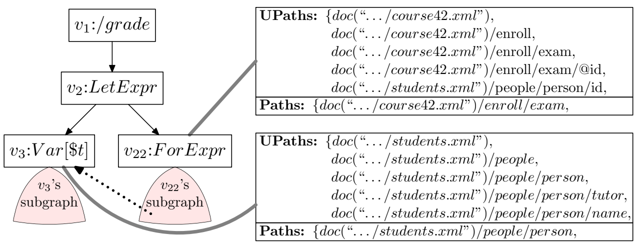
Excerpt of request message for makenodes()

```
⟨request⟩
  ⟨projection-paths⟩
   ⟨used-path/⟩
   ⟨returned-path⟩parent::a⟨/xrpc:returned-path⟩
  ⟨/projection-paths⟩
  ⟨fragments/⟩
```

Excerpt of response message for makenodes()

```
⟨env:Envelope . . . ⟩
  ⟨env:Body⟩
   ⟨response⟩
    ⟨fragments⟩ ⟨fragment⟩⟨a⟩⟨b⟩⟨c/⟩⟨/b⟩⟨/a⟩⟨/fragment⟩ ⟨/fragments⟩
    ⟨call⟩ ⟨sequence⟩⟨element fragid="1" nodeid="2"⟩⟨/sequence⟩ ⟨/call⟩
   ⟨/request⟩
  ⟨/env:Body⟩
⟨/env:Envelope⟩
```

Fig. 5. Pass-By-Projection Messages



UPaths: {doc("…/$course42.xml$"),
doc("…/$course42.xml$")/enroll,
doc("…/$course42.xml$")/enroll/exam,
doc("…/$course42.xml$")/enroll/exam/@id,
doc("…/$students.xml$")/people/person/id,
Paths: {doc("…/$course42.xml$")/$enroll/exam$,

UPaths: {doc("…/$students.xml$"),
doc("…/$students.xml$")/$people$,
doc("…/$students.xml$")/$people/person$,
doc("…/$students.xml$")/$people/person/tutor$,
doc("…/$students.xml$")/$people/person/name$,
Paths: {doc("…/$students.xml$")/$people/person$,

Fig. 6. Path annotation example

upper part of Figure 5 shows part of the request message for the call from Q_1 (discussed in Problem 4):

let $bc := execute at {"example.org"} {makenodes()}

since the projection path analysis detects that $bc *will subsequently be used as context node by a parent step:* $abc := $bc/parent::a, *the request message specifies* parent::a *as a returned path. Therefore, the response message contains the full fragment $\langle a\rangle\langle b\rangle\langle c/\rangle\langle/b\rangle\langle/a\rangle$ to which* $abc *then gets correctly bound.*

A. Extending Projected XML

We extend the path grammar rules [21] and path annotations, to handle full-fledged XQUERY involving reverse/horizontal XPATH steps and built-in functions. The extended grammar rule for *ProjectionPath* is given in Table V.

We denote path annotations in projected XML as follows:

$$Env(v_i) \vdash Expr \Rightarrow Paths_1 \text{ using } Paths_2$$

The notation $Env(v_i)$ is used to identify the path annotation environment at a certain vertex v_i in the XQUERY *d*-graph.

Path annotations are constructed bottom up by *path analysis rules* that derive the set of used ($UPaths$) and returned ($Paths$) paths for each XCORE expression in terms of used and returned paths of its subexpressions. Therefore, we extend the notation of the vertices and use $v_i.UPaths$ and $v_i.Paths$ to refer to the path sets, with which the vertex v_i is annotated.

Example 6.2: *Assume that the subgraph $G_{v_{22}}$ rooted at v_{22} in Figure 2 is identified to be evaluated remotely. The subgraph $G_{v_{22}}$ has one parameter, $\$t$, via the* VarRef *edge (v_{36}, v_3). We show the path annotations of v_3 and v_{22} in Figure 6. Comparing the returned path of v_3 with all projection paths of v_{22}, we know that only the* id *child elements of the* person *elements are needed. Thus, only those elements will be projected and serialised in the request message for v_{22}. For reasons of space, document* URIs *are abbreviated, and the annotations of v_1 and v_2 are omitted, which in reality also have such annotations.*

The basic path analysis rules have been discussed in [21], such as *literal values*, *sequences*, for and let expressions and XPATH steps, etc. Our extension to include reverse and horizontal XPATH steps brings no changes for the path analysis rules, but must be supported by the loading algorithm, which is described in Section VI-B. We complement the rules for built-in functions, which apart from the unsolved cases mentioned under *Problem 5* in Section II (fn:root(), fn:id(), fn:idref() and fn:lang()) also includes fn:doc(). The description of the basic projection technique assumes a single document. As in distributed query processing there are always multiple documents, our paths always start with fn:doc(URI).

Path Analysis Rules. We provide one rule for fn:doc() with a constant parameter and another for computed URIs:

$$\frac{()}{Env(v_i) \vdash \text{fn:doc}(Literal_1) \Rightarrow \text{fn:doc}(Literal_1{::}v_i) \text{ using } \emptyset} \quad (\text{DOC}_1)$$

$$\frac{Env(v_j) \vdash Expr_j \Rightarrow Paths_j \text{ using } UPaths_j}{\begin{array}{l}Env(v_i) \vdash \text{fn:doc}(Expr_j) \Rightarrow \text{fn:doc}(*{::}v_i) \text{ using} \\ \qquad Paths_j \cup UPaths_j \cup Paths_j/\text{descendant::text()}\end{array}} \quad (\text{DOC}_2)$$

As mentioned in Section IV, in the definition of $D(v_x)$[5], we use a wildcard URI$*$ if the document name is an expression. All paths start with doc(uri::v_i), thus, they identify both document URI and the vertex v_i where it is loaded. This notation facilitates the identification of situations where the same URI is loaded twice (the function hasMatchingDoc()). A similar rule can be formulated for XML element construction, producing a return path doc($v_i :: v_i$) with an artificial unique URI. Also note that XQUERY implicitly converts the actually values of function parameters, thus, in all rules in this section, we add a descendant::text() step to each returned path of a parameter. The rule for fn:root() is:

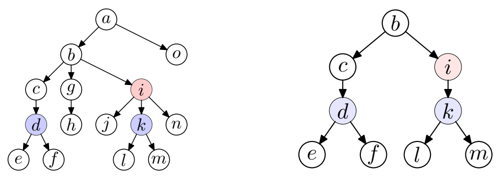
$$\frac{Env(v_j) \vdash Expr_j \Rightarrow Paths_j \text{ using } UPaths_j}{Env(v_i) \vdash \text{fn:root}(Expr_j) \Rightarrow \cup_{p \in Paths_j} p/\text{root() using } UPaths_j} \quad (\text{ROOT})$$

The built-in function fn:root() with a single parameter is treated in the path annotations much like XPATH axis steps, where the parameter has become the path prefix. In this path notation, functions remain easily recognisable by the parentheses. The rules for the built-in functions fn:id()/fn:idref(), are highly similar (only fn:id() provided):

$$\frac{\begin{array}{l}Env(v_j) \vdash Expr_j \Rightarrow Paths_j \text{ using } UPaths_j \\ Env(v_k) \vdash Expr_k \Rightarrow Paths_k \text{ using } UPaths_k\end{array}}{\begin{array}{l}Env(v_i) \vdash \text{fn:id}(Expr_j, Expr_k) \Rightarrow \cup_{p \in Paths_k} p/\text{id() using} \\ \qquad Paths_j \cup UPaths_j \cup Paths_j/\text{descendant::text()} \cup Paths_k \cup UPaths_k\end{array}} \quad (\text{ID})$$

The first parameter of fn:id() is ignored by the annotations, as it contains string values, and the annotation framework only allows for the estimation of node sets. This has the consequence that our loading algorithm will conserve *all*

[5]We use the doc(..) prefixes of the *returned paths* annotations on v as a more precise form of the $D(v)$ property. Documents that were only used but not returned will also be part of the original $D(v)$, but these will not cause semantic problems.



(a) Original XML tree $\mathcal{D}$ (b) Projected tree $\mathcal{D}'$

Fig. 7. Runtime XML Projection Example

elements with an ID/IDREF attribute. Finally, the rule for fn:lang() is:

$$\frac{\begin{array}{l} Env(v_j) \vdash Expr_j \Rightarrow Paths_j \text{ using } UPaths_j \\ Env(v_k) \vdash Expr_k \Rightarrow Paths_k \text{ using } UPaths_k \end{array}}{\begin{array}{l} Env(v_i) \vdash \text{fn:lang}(Expr_j, Expr_k) \Rightarrow \emptyset \text{ using } Paths_j \cup UPaths_j \cup \\ \quad Paths_j/\text{descendant::text()} \cup Paths_k \cup UPaths_k \cup \\ \quad Paths_k/\text{ancestor::}* \cup Paths_k/\text{ancestor-or-self::}*/\text{attribute::xml:lang} \end{array}} \quad (\text{LANG})$$

The built-in function fn:lang() tests whether the language of its node parameter $Expr_k$, as specified by xml:lang attributes, is the same as (or is a sublanguage of) the language specified by its string parameter $Expr_j$. The language of $Expr_k$ is determined by the value of the XPATH expression: (ancestor-or-self::∗/attribute::xml:lang)[last()]. All paths are propagated as used paths, as this function returns a boolean value.

B. Runtime XML Projection

The extensions we made to XML projection, namely support for reverse/horizontal XPATH axes and fn:root(), fn:id(), fn:idref() and fn:lang(), could not be trivially integrated in the loading algorithm of [21]. However, in this case we are not really looking for a loading algorithm that efficiently reads (shreds) an XML file into a projected representation. Rather, the documents are already present (and indexed) in the XQUERY engine, and runtime message projection is a *serialisation* task. Therefore, we propose a new *runtime* approach for projection, targeted at serialisation, rather than at shredding. Whereas the original loading algorithm starts at the document root, and evaluates *absolute* used and returned paths, our runtime projection algorithm starts in a run-time state, that is, with a real, *materialised context sequence* (e.g., the parameter values that are about to be serialised in a SOAP message), and executes only *relative* paths on them. Because the node sequence bound at run-time to a function parameter is only a subset of the node set characterised by its compile-time path annotation (e.g., its contents may well have been reduced by applying a selection predicate), this runtime projection technique can be much more precise than the original projection algorithm. As a final consideration, the projected XRPC messages trade projection effort for network bandwidth, which especially in WAN scenarios plays in the advantage of projection.

For these reasons, our *runtime* approach for projection simply relies on the normal XPATH evaluation capabilities of the XQUERY engine for fully evaluating all used and returned path annotations one-by-one (and uniting them with union()). Doing so, it produces a *used node set U* and a *returned node set R*. These two sets are the input for the runtime projection algorithm listed in Algorithm 1.

The Runtime Projection Algorithm identifies all projection nodes in the XML tree representation of the original document, by traversing the tree top-down depth-first. During traversal, if

```
Algorithm 1: RUNTIMEXMLPROJECTION ($U, R, \mathcal{D}$)
  input:  $U$- used nodes
          $R$- returned nodes
          $\mathcal{D}$- the original XML document
  output: $\mathcal{D}'$- the projection of $U$ and $R$ on $\mathcal{D}$
1  projection nodes $P \leftarrow$ sort($U \cup R$);
                              ▷$P$ is union of $U$ and $R$ sorted by document order
2  $proj \leftarrow$ first node in $P$;
3  $cur \leftarrow$ first node of $\mathcal{D}$, i.e., root node;
4  while $\neg P.end()$ do
5      if $proj$ is a descendant of $cur$ then
6          add $cur$ to $\mathcal{D}'$;
7          $cur \leftarrow$ next node in $\mathcal{D}$;
8      else if $proj = cur$ then
9          if $proj$ is a returned node then
10             add $cur$ and all descendants of $cur$ to $\mathcal{D}'$;
11             $cur \leftarrow$ next following node of $cur$ in $\mathcal{D}$;
12             while $proj.next$ is a descendant of $proj$ do
13                 $proj \leftarrow proj.next$;          ▷ prune projection nodes;
14             end
15         else
16             add $cur$ to $\mathcal{D}'$;
17             $cur \leftarrow$ next node in $\mathcal{D}$;
18         end
19         $proj \leftarrow proj.next$;                  ▷ next projection node;
20     else
21         $cur \leftarrow$ next following node of $cur$ in $\mathcal{D}$;
22     end
23 end
24 $cur \leftarrow$ root node of $\mathcal{D}'$;
25 while $cur$ has only one child node $\wedge$ $cur \notin \{U \cup R\}$ do
26     $cur \leftarrow$ first child of $cur$;
27 end
```

the current node cur of the XML document is an ancestor of the current projection node $proj$ (line 5), cur is added to output $\mathcal{D}'$ and moved to the *next node* in document order. If a $proj$ is found (line 8), $proj$ is added to $\mathcal{D}'$; if this $proj$ is a returned node, all its descendants are also appended. Then cur is moved to its *next following node* in the document. Otherwise, if the current projection node $proj$ is not a descendant of cur, the subtree of cur can be skipped (line 21). Though this algorithm is formulated on an abstract level that is independent of the particular XML storage scheme used in an XQUERY engine, it is safe to assume that skipping a subtree is fast (either $O(1)$ or $O(log(|\mathcal{D}|))$). At the end of the algorithm (lines 24-27), post-processing is performed to remove unnecessary nodes, as we are only interested in the *lowest common ancestor* of all input nodes in the projected document $\mathcal{D}'$.

Example 6.3: *Consider an* XML *document $\mathcal{D}$ in Figure 7(a). Assume that the used node set U is $\{i\}$, and the returned node set R is $\{d, k\}$. Figure 7(b) shows the projected document $\mathcal{D}'$ of applying Algorithm 1 on U, R and $\mathcal{D}$.*

The algorithm starts with $P \leftarrow \{d, i, k\}$, $proj \leftarrow d$ and $cur \leftarrow a$. We traverse the tree using cur from a to d. Nodes a, b and c are added to $\mathcal{D}'$, since they are ancestors of the current context node d. Nodes d, e and f are also added to $\mathcal{D}'$, as d is a returned node. Then, cur is advanced to g (d's next following node). Because the next context node i is not in the subtree of g, the subtree is skipped by advancing cur to i. Recall that i is a used node, thus only i is added to $\mathcal{D}'$. The last context node is k. Our current document node cur traverses from i to j, and then to k, where we can add nodes k, l and m to $\mathcal{D}'$. The traversal can be terminated, because there is no more context nodes to process. However, the intermediate result $\mathcal{D}'$ contains all *common ancestors of $\{d, i, k\}$. The post-processing removes node a from $\mathcal{D}'$, which produces the final projected document $\mathcal{D}'$ as shown in Figure 7(b).*

Relative projection paths. At *compile time*, the XQUERY compiler builds a query graph (d-graph) with root v_{root}, normalises it followed by decomposition and code motion.

For each inserted XRPCExpr v_{xrpc}, and for each XRPCParam parameter vertex v_{param}, it then extracts the relative paths:

$U_{rel}(v_{param})$ = allSuffixes($R(v_{param}), U(v_{xrpc})$)
$R_{rel}(v_{param})$ = allSuffixes($R(v_{param}), R(v_{xrpc})$)
$U_{rel}(v_{xrpc})$ = allSuffixes($R(v_{xrpc}), U(v_{root})$)
$R_{rel}(v_{xrpc})$ = allSuffixes($R(v_{xrpc}), R(v_{root})$), with:
allSuffixes($Paths_i, Paths_j$)= $\{s_j | p_i/s_j \in Paths_j : \exists p_i \in Paths_i\}$

At *runtime*, $\cup_{\forall v_{param}} U_{rel}(v_{param})$ and $\cup_{\forall v_{param}} R_{rel}(v_{param})$ are used to project the parameters in the outgoing XRPC request message. $U_{rel}(v_{xrpc})$ and $R_{rel}(v_{xrpc})$ are passed in the projection-paths element such that a remote peer can appropriately apply these paths to project the response message.

Projecting a document with Algorithm 1 requires pre-calculated used and returned node sets. These sets are simply computed using the XPATH evaluation infrastructure of the underlying XQUERY engine, by feeding the intermediate result \$ctx$_{param}$ corresponding to v_{param} as context sequence into all suffix paths $s_i \in U_{rel}(v_{param})$ (resp. $R_{rel}(v_{param})$):

union(\$ctx$_{param}/s_1$, union(\$ctx$_{param}/s_2$,...
union(\$ctx$_{param}/s_{n-1}$, \$ctx$_{param}/s_n$)...))

Paths \$ctx$/path_i/$root()$/path_j$ with function fn:root() are executed as root(\$ctx)$/path_j$. Similarly, \$ctx$/path_i/$id()$/path_j$ is executed as root(\$ctx)//attribute()::$(a_1|..|a_n)/../path_j$, where $a_1,..,a_n$ are all ID attributes[6] (resp. IDREF in case of idref()).

The request handler on the remote side uses the same method to evaluate the suffix paths $U_{rel}(v_{xrpc})$ and $R_{rel}(v_{xrpc})$ using the result sequence of the function as \$ctx$_{xrpc}$ during serialisation of the response message.

Interoperability. We have devised a way to support pass-by-projection in the XRPC WRAPPER by substituting the projection algorithm with a variant that serialises the lowest common ancestor of the used and returned node sets. Since document projection is not expressible in XQUERY (not even with the TRANSFORM feature of XQUF), this is as far as a pure XQUERY engine can get. We contemplate the possibility to let the XRPC WRAPPER echo the SOAP response message it generates to a stream, and implement a streaming version of our projection algorithm (that first gets a stream of used and returned nodes, and then the to be projected fragments) inside the XRPC WRAPPER java program. In case of XML data with a user-defined XML SCHEMA, the default projection algorithm is likely to throw away mandatory elements and attributes. For this reason, the runtime projection algorithm should be made schema-aware. A simple solution is to ensure that only elements with a minoccurs declaration of zero (i.e., optional elements) are removed. One can also envision more advanced variants that further reduce the size of a typed XML document.

VII. Decomposition of XQUF Queries

Since the introduction of the W3C XQUF [7] specification, which has been well-received and adopted by various XQUERY engines (e.g., [9], [11], [16], [23], [26], [27], [36]), XQUERY is no longer a read-only query language. We now show how we can leverage such update-capable XQUERY engines to automatically rewrite purely local updates into queries that may push some computations to remote peers. We recall that the general processing model of XQUF is that first the read-only part of a query is executed that defines which nodes are going to be updated, and how. This first phase results in a *pending update list* (PUL). In the second phase all update actions in this list are executed. Therefore, the first phase of XQUF execution is identical to a read-only query, and can in principle be distributed in the same way as described in the previous sections. However, systems implementing the XQUF typically only allow to update persistently stored documents, e.g., updating documents on an HTTP URI is not allowed. In this section, we first explain the restrictions the XQUF imposes on XRPC query distribution. Then, we extend the semantics of XQUF to allow updates on documents opened with fn:doc() using xrpc://$\mathcal{P}$/$\mathcal{D}$ URIs (in short: *remote documents*) and also support the fn:put() XQUF built-in function to write entire new documents to such URIs. This extended semantics again creates a possible trade-off between data shipping vs. function shipping, namely retrieving and updating a local copy of a remote document followed by an fn:put() vs. executing an XQUF updating function over XRPC. We introduce the necessary constraints to our query distribution techniques that guarantee semantic equivalence for such queries.

A. Distributing Normal XQUF Queries

XQUF has extended the XQUERY language with four kinds of *updating expressions*: UpdExpr = {InsertExpr, DeleteExpr, RenameExpr, ReplaceExpr} (Table II). An XCORE query containing at least one UpdExpr is an *updating* XCORE *query* (in short: updating query). Each UpdExpr has a TargetExpr that identifies the *target nodes* to be updated, and (except for DeleteExpr) each has an ExprSingle that computes the new values. For simplicity, we refer to those ExprSingle as SrcExpr, although XQUF uses different names. The functionality of the first three kinds of expressions is self explanatory. With ReplaceExpr, one can replace the target node with a new sequence of nodes ("replace node"), or replace the value of the target node ("replace value"). The expressions RenameExpr and "replace value" only modify some properties of the target node without changing its *node identity*.

XQUF also defines a *transform expression* (TransformExpr) that creates (and possibly modifies) copies of existing XML nodes. Each node created by a TransformExpr has a new node identity. The result of a TransformExpr is an XDM (XQuery Data Model) instance that may include both new nodes created by the TransformExpr and existing nodes. TransformExpr has special semantics: it is *not* an updating expression, as it does not modify any existing nodes. Hence, an XCORE query that merely contains UpdExpr as subexpressions of a TransformExpr is *not* an updating query.

In our XCORE rewriting framework, all three algorithms use a by-value based semantics, which means that target nodes may not stem from an XRPC function result, or from a function parameter (if the updating expression occurs inside an XRPC function body). Hence, we enforce that all UpdExprs, denoted V_u, must be executed on the same peer that opened the document using fn:doc(). This, in turn, enforces that all expressions V_{a_i} (except TransformExpr), which depend on a $v_{u_i} \in V_u$,

[6]Note that these a_i should be determined at runtime by the XRPC projection algorithm. The impossibility to express selection of all ID/IDREF attributes in XQUERY, and thus in the XRPC WRAPPER, forces us to still avoid shipping expressions where the result of v_{xrpc} is used as input to id()/idref().

must be executed on the local peer. This is because V_{a_i} could only parse-depend on a v_{u_i}, as updating expressions are not allowed in a variable binding. Decomposing an expression in V_{a_i} would cause the v_{u_i} to be executed on a remote peer. To correctly identify the target nodes of an UpdExpr, all expressions V_{t_i} that produce target nodes for a v_{u_i}, must also be executed on the local peer. When decomposing an updating query, the vertices V_u, V_{t_i}, and V_{a_i} in the query's *d*-graph are never valid decomposition points, regardless of the parameter passing semantics used by the decomposition algorithm. The following XQUF *insertion conditions* should be added to the insertion conditions of each decomposition algorithm.

XQUF Insertion Conditions. Given a *d*-graph G and a subgraph G_{r_s} of G rooted at vertex r_s, under any semantics, r_s is in the set $I(G)$ of valid decomposition points, iff r_s *also* satisfies all of the following conditions:

a. $r_s.rule \notin \{\text{UpdExpr, TargetExpr}\}$

b. $\nexists v_u \in V(G) : v_u.rule \in \{\text{UpdExpr}\} \wedge r_s.rule \neq \text{TransformExpr} \wedge r_s \rightsquigarrow v_u \wedge (\nexists v_m \in V(G) : v_m.rule = \text{TransformExpr} \wedge r_s \rightsquigarrow v_m \rightsquigarrow v_u)$

c. $\nexists v_t \in V(G) : v_t.rule = \text{TargetExpr} \wedge v_t \rightsquigarrow r_s \wedge (\exists p_t \in v_t.Paths \wedge \exists p_s \in r_s.Paths \wedge \text{starts-with}(p_t, p_s)$

Condition *a* avoids decomposing any UpdExpr and TargetExpr. Condition *b* states that if r_s is not a TransformExpr, r_s may not depend on an UpdExpr, unless the UpdExpr is a subexpression of a TransformExpr, on which r_s depends. Condition *c* states that r_s may not be decomposed, if r_s produces target nodes of an UpdExpr. We say r_s *produces target nodes*, iff a returned path p_s of r_s is a prefix of a returned path p_t of v_t, i.e., nodes returned by r_s include target nodes. Note that condition *b* allows a TransformExpr to be decomposed by all three decomposition algorithms, as it always makes (deep) copies of its source nodes. If a TransformExpr is executed on peer $\mathcal{P}$, $\mathcal{P}$ becomes the "local peer" for all new nodes created by this TransformExpr. With condition *a*, we prevent UpdExprs in the modify clause of a TransformExpr from being separated from the TransformExpr (i.e., executed on another peer than $\mathcal{P}$). Thus, the UpdExprs in the modify clause will also be executed on $\mathcal{P}$, which is the local peer of their target nodes. This confirms the XQUF semantics that UpdExprs may only be applied to local nodes. In the remainder of this section, we continue our discussion on processing UpdExprs that are not subexpressions of a TransformExpr.

B. Updating XCore Queries on Remote Documents

We now extend the semantics of XQUF to allow updates on remote documents (i.e., documents identified by an xrpc:// URI scheme). We first provide a semantics for such updates in normal non-distributed execution (i.e. data shipping): the read-only part of the query is evaluated first, retrieving (a copy of) all accessed remote documents to the local peer, which results in a PUL. Then, the standard XQUF function upd:applyUpdates() is executed to carry through all update actions in the PUL. This could modify (some of) the local copies of the remote documents. Finally, as an additional step, for each affected remote document, an fn:put() is executed by passing the document's original URI and its new contents, effectively replacing the existing document on the remote peer with the modified one. This semantics does not apply to XCORE queries only containing transform expressions, as they are read-only queries. Thus, no additional fn:put() is executed to overwrite the existing documents.

Formal Semantics. Let Q_u denote an XCORE query containing at least one UpdExpr on a remote document and G_u its *d*-graph. $D_u(Q_u)$ denotes the set of *affected documents* that may be updated by Q_u:

$$\begin{aligned} D_u(Q_u) = \{(uri,)|\exists v_u, v_y, v_z \in V(G_u) : \{v_y, v_z\} \in E(G_u) \wedge \\ uri = v_z.val \wedge node = \text{doc}(uri) \wedge v_y.rule = \text{FunApp} \wedge \\ v_y.val = \text{“doc”} \wedge v_z.rule = \text{Literal} \wedge v_u.rule = \text{TargetExpr} \wedge \\ uri = v_z.val \wedge \exists p \in v_u.Paths \wedge \text{starts-with}(p, uri)\} \end{aligned}$$

$D^r_u(Q_u)$ is a subset of $D_u(Q_u)$, which contains the *affected remote documents*: $\forall d^r_i \in D^r_u(Q_u)$: starts-with$(d^r_i.uri,$ "xrpc://"). The auxiliary functions host() and path() extract, from an XRPC URI "xrpc://$\mathcal{P}$/$\mathcal{D}$", the peer identifier $\mathcal{P}$ and the document name $\mathcal{D}$, respectively. Each query operates in a *database state* (db^p), which includes the documents and their contents persistently stored in the XML database on p. The $dynEnv.docValue$ from [10] corresponds to db^p used here.

The formal semantics of distributed updates is[7]:

$$\frac{\begin{array}{l} \forall d^r_x \in D^r_u(Q_u) : \text{fn:doc}(d^r_x.uri) \Rightarrow D^{r'}_u(Q_u) \\ db^{p_0}, D^{r'}_u(Q_u) \vdash Q_u \Rightarrow \Delta; \\ db^{p_0}, D^{r'}_u(Q_u) \vdash \text{upd:applyUpdates}(\Delta) \Rightarrow db'^{p_0}, D^{r''}_u(Q_u); \\ (R^u) \quad \forall d^{r''}_x \in D^{r''}_u(Q_u) : \text{fn:put}(d^{r''}_x.node, d^{r''}_x.uri) \Rightarrow (), db^{\text{host}(d^{r''}_x.uri)}; \end{array}}{db^{p_0} \vdash Q_u \Rightarrow (), db^{p_0}}$$

The rule R^v states that the execution of an updating query Q_u at the *local peer* p_0 in the database state db^{p_0} starts with retrieving the remote documents $D^r_u(Q_u)$, which could potentially be affected by Q_u, to p_0[8]. This yields a set of *local copies* $D^{r'}_u(Q_u)$ of $D^r_u(Q_u)$. Note that this step does not change db^{p_0}, as the documents in $D^{r'}_u(Q_u)$ are transient documents. Then, Q_u is executed in db^{p_0} with the additional documents $D^{r'}_u(Q_u)$, which first yields a PUL Δ. Subsequently, upd:applyUpdates() is executed to apply all update primitives in Δ to the affected documents. Updates in Δ that should be applied on remote documents $D^r_u(Q_u)$ are applied on their local copies $D^{r'}_u(Q_u)$ instead. This step produces a new current database state db'^{p_0}, which could differ from db^{p_0} (if Δ contains updates on really local documents), and a set of changed local copies $D^{r''}_u(Q_u)$. Finally, an additional step is executed, which calls fn:put() to store each $d^{r''}_i \in D^{r''}_u(Q_u)$ on its hosting peer and overwrite the existing $d^r_i \in D^r_u(Q_u)$. This step also creates a new current remote database state $db^{host(d^{r''}_x.uri)}$ on each hosting peer. As the rule R^u only applies Δ at the end of query execution, updates are not visible for the same query, which confirms the XQUF semantics. Hence, if Δ only contains updates on a single document, this rule already provides atomic updates.

Isolation Levels. Note that the - potentially multiple - fn:put("xrpc://..") together with potential updates on some local documents constitute a *distributed updating query*. Depending

[7]We use the ':' sign to suggest an order in the evaluation of the premises.

[8]As explained in Section IV, computed URIs and invocations of fn:collection() are represented by $*$. During the runtime, when the actually value of the wildcard symbols are available, more URIs might be added to the set $D^r_u(Q_u)$ on the fly.

on the semantics desired by the user, this distributed updating query could be run in a certain consistency level, which has been discussed in detail in our previous work [38]. One option is *no consistency* at all, in which some documents may get updated, but other document updates may fail or get lost. By tagging queries with a unique ID, the *repeatable read* consistency level can be easily achieved. To ensure distributed atomic updates, [38], [39] shows how the WS-AtomicTransaction standard [35] can be integrated into XRPC to provide 2PC. In addition to repeatable reads and atomic commits, the *lost updates* anomaly can be avoided if participating peers abort the 2PC commit when another updating query or fn:put() has modified an updated document already. Note that these semantics can also be supported by the XRPC WRAPPER if the XQUERY engine is XRPC oblivious. Given the design goal for XRPC of supporting P2P applications on the Internet, we refrained from attempting to define higher consistency levels (e.g. distributed serialisability), as the overhead of these are impractical in such environments. We consider more advanced distributed consistency levels for P2P on the Internet a topic of future work, and consider it out of scope here, where we focus on semantically correct distributed query rewriting.

XQUF Rewrites. In principle, we cannot push any UpdExprs, except *homogeneous updating expressions*. An UpdExpr v_u^h is *homogeneous*, iff all returned paths of its TargetExpr $v_{t_u}^h$ start with the same "xrpc://$\mathcal{P}$", i.e., the update affects only nodes that stem from a single peer, hence the update can be pushed to that peer using an XQUF *updating function* such that it acts only on local documents there. The insertion conditions for updates formulated in Section VII-A also applies for pushed updating expression: target nodes of an UpdExpr v_u may not be passed to a remote peer as function parameters or results. Decomposition of v_u^h thus requires that all expressions $V_{t_u}^h$ that produce target nodes of v_u^h must be executed in the same remote function as v_u^h. So, we need to find the smallest (super-)expression v_s that contains both v_u^h and $V_{t_u}^h$.

Let Q_u be an updating query containing the homogeneous UpdExpr v_u^h and G_u its *d*-graph. Let v_w^h be the TargetExpr of v_u^h (i.e.: $(v_u^h, v_w^h) \in E(G_u) \wedge v_w^h.rule = \text{TargetExpr}$). We define $V_{t_u}^h$ as:

$$\forall v_i \in V_{t_u}^h : v_w^h \rightsquigarrow v_i \wedge$$
$$(\forall p_t \in v_i.Paths, \forall p_w \in v_w^h.paths : \text{starts-with}(p_w, p_t))$$

and define v_s as:

$$(v_s \rightsquigarrow v_u^h \vee v_s = v_u^h) \wedge \forall v_i \in V_{t_u}^h : v_s \rightsquigarrow v_i \wedge$$
$$\nexists v_x \in V(G_u) : v_s \rightsquigarrow v_x \wedge v_x \rightsquigarrow v_u^h \wedge \forall v_i \in V_{t_u}^h : v_x \rightsquigarrow v_i$$

Then, v_s could be a valid decomposition point. If no such point can be found, we fall back to the data shipping strategy (i.e., local execution and a fn:put("xrpc://$\mathcal{P}$/$\mathcal{D}$") at the end of query execution). For updating queries containing both push-able and not push-able UpdExprs, however, there is an additional issue to deal with: we can only push an UpdExpr v_{u_0} if we can guarantee that no other UpdExprs elsewhere in the query update nodes from the same documents (a *clash*), *or*, if another UpdExpr does, it can also be pushed. This is because all UpdExprs that are not pushed will generate an fn:put() in the end, which would potentially overwrite the pushed updating actions or other fn:put()s, from the same transaction. However, if all updates to the same document are pushed, the 2PC protocol used in XRPC ensures correct execution [38].

VIII. Evaluation in MonetDB/XQuery

We have implemented the proposed algorithms in MonetDB/XQuery [4], a purely relational XDBMS that uses the *Pathfinder* [14] XQUERY compiler. We use the XRPC extension for remote function evaluation. Note that, as there are no other comparative results exist, the main goal of our experiments is to show the impact of the proposed techniques in a step-by-step fashion.

A. Read-only Queries

For the all our experiments, the test platform consisted of three 2GHz Athlon64 Linux machines connected in a local network (LAN). Each was equipped with a 2GB RAM. The benchmark data used is XMark [29], a popular XML benchmark for evaluating XQUERY efficiency and scalability. The data set was generated using scalar factors 0.1, 0.2, 0.4, 0.8 and 1.6. A data set is stored on each remote peer. We conducted three groups of experiments: bandwidth usage, query execution time and runtime projection precision.

We slightly modified the query Q_2^n (in Table III) so that it conforms to the XMark schema as the following:

```
(let $t := let $s := doc("xrpc://peer1/xmk_nn_MB.xml")/child::site
                              /child::people/child::person
              return for $x in $s return
                          if ($x/descendant::age < 40) then $x else ()
 return for $e in (let $c := doc("xrpc://peer2/xmk_nn_MB.auctions.xml")
                          return $c/descendant::open_auction)
             return if($c/child::seller/attribute::person = $t/attribute::id)
                        then $c/child::annotation else () )/child::author
```

All techniques discussed in this paper are applied on the above query: *(i)* under the pass-by-value semantics, only the expression "doc('xrpc://peer1/xmk_nn_MB.xml')/. . . /child::person" can be decomposed and executed on peer1; *(ii)* under the pass-by-fragment semantics, we can decompose both the second let clause ("let $s := ...") and the second for-loop ("for $e in ..."), and execute them on peer1 and peer2 respectively. The variable $t becomes the parameter of the generated function containing the second for-loop (see also Table IV); *(iii)* under the pass-by-projection semantics, the query is decomposed in the same way as using pass-by-fragment, however, when serialising the request messages, a projection of $t/attribute::id (parameter projection) and $c/child::annotation/child:author (result projection) is calculated. The test set thus contains four queries in total, and each of them is executed on 2 documents of sizes 10, 20, 40, 80 and 160MB.

In this case, code motion is ideal, as it is able to send just strings, not nodes. However, if we would replace the final step child::author by parent::$*$, then just applying code motion and no projection provides mediocre performance similar to by-value. It is the ability of projection to decompose almost any query at little cost, that makes it the overall method of choice.

Bandwidth Usage. Figure 8 shows the bandwidth used by each benchmark query on different set of documents, i.e., the total size of XML documents plus total size of XML messages transferred among peers, in its y-axis. The x-axis is the total size of the XML documents used by each query. The pure data-shipping XQUERY query (the left most bar) costs the largest bandwidth usage, as both documents used by the query have

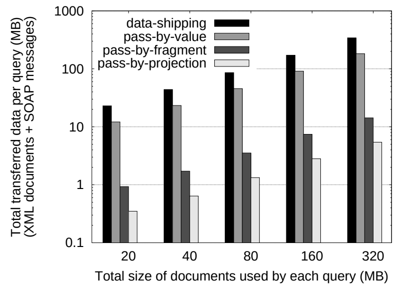


Fig. 8. Bandwidth Usage

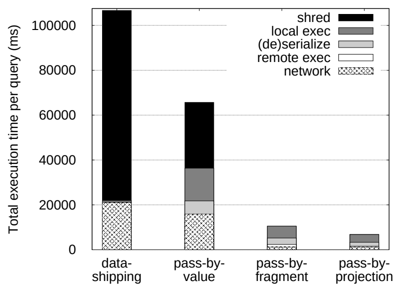


Fig. 9. Time Breakdown (320MB data)

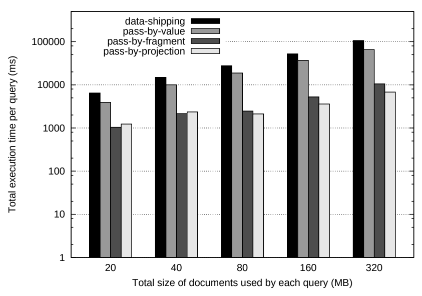


Fig. 10. Execution Time

to be shipped. By-value decomposition can push the XPATH step doc(“xrpc://peer1/xmk_nn_MB.xml”)/. . . /child::person to be evaluated on peer1, which reduces the amount of data sent from peer1 to the local peer. However, the second document “xmk_nn_MB.auctions.xml” still has to be sent fully. The by-fragment passing semantics allows to push predicates to both peers, achieving a distributed semijoin plan. Also, it strongly reduces message size by avoiding duplicating the same XML node multiple times. Pass-by-projection further brings down message sizes due to reduced response message size. For example, when sending the result of remote execution of the second for-loop, the response message will only contain annotation nodes with their author child nodes. When applied on pass-by-fragment, code motion has larger effect in reducing message size than it is applied on pass-by-projection. This is because in pass-by-fragment, complete person nodes (i.e., including all their descendants) are serialised, while in pass-by-fragment with code motion, only the values of the id attributes are serialised. In pass-by-projection, however, the message size has already minimised the data to be sent, i.e., only person nodes and their id attributes, hence, the effect of applying code motion here is negligible. In general, we observe good scalability of pass-by-fragment and pass-by-projection in bandwidth usage.

Execution Time. Figure 9 shows the execution time breakdown of all four queries on documents of 320MB in total. The execution time is divided into five parts: *shred* is the time to receive a document from the remote peer and shred it in to the XML database; *local exec* is the execution time of the query at local peer, including query parsing, module loading, etc; *(de)serialise* is the time spent on generating/shredding the XML messages and extracting parameter/result values from the messages; *remote exec* is the time to execute the called functions on remote peers; and *network* is the time spent on sending/receiving the XML messages. From Figure 9, the following observations can be made: (i) in the data-shipping only query and the by-value decomposed query, data shredding is the main bottleneck, either because the whole document will be shipped (data-shipping), or an XML node might be shredded multiple times (by-value). Especially in the data-shipping query, more than 99% of the total execution time is spent on getting the documents from remote peers and shredding them; (ii) when pass-by-fragment and pass-by-projection semantics are used, the total execution time is significantly reduced (about $84 \sim 94\%$, comparing with data-shipping and pass-by-value). This is easily explained as these techniques reduce the amount of data exchanged to be less than 10% of the original document sizes. Even with the overhead introduced by remote execution (i.e., ‘*(de)serialise*’$+$‘*remote exec*’), pass-by-fragment or pass-by-projection are preferred over the data-shipping method. (iii) pass-by-projection performs even better than pass-by-fragment (about 35% improvement), which is again explained by the reduced bandwidth usage, as shown in Figure 8.

Figure 10 shows the execution time of all queries on documents of increasing sizes, which indicates that the two enhanced parameter passing techniques achieve good scalability. On average, pass-by-fragment and pass-by-projection achieve a performance improvement of roughly 94%, comparing with data-shipping; this is proportional to the decreasing in bandwidth usage, which is approximately 96%. Even on small documents (20MB), the proposed techniques are preferred over the data-shipping methods.

Runtime projection precision. Our new runtime projection technique combines intermediate query results with runtime execution or relative XPATH paths. Due to selections (by e.g., predicates and value comparisons), the run-time projection node sets obtained may be much smaller than suggested by compile-time projection paths, used in [21]. We used our by-projection benchmark query to compare runtime projection with compile-time projection, on various sizes of the XMark document “xmk_nn_MB.xml”. In this experiment, the compile-time technique projects all person elements and their age, while our runtime projection technique will only project those person elements that have an age descendant larger than 45. Figure 11 shows runtime projection to be 5 times more precise in terms of the size of projected document. In the case of this experiment, the investment in run-time XPATH evaluation pays off due to the more precise results, as shown in Figure 12.

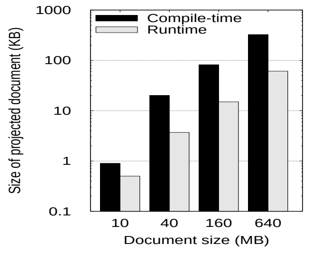


Fig. 11. Selected Nodes

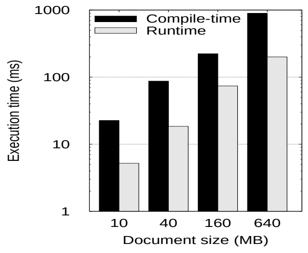


Fig. 12. Execution Time (ms)

B. XQUF Queries

For the updating XCORE queries, we have conducted two groups of experiments to compare performance of updating remote documents with or without XRPC. The first group

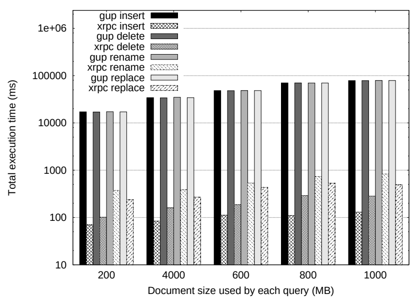


Fig. 13. Updates Execution Time

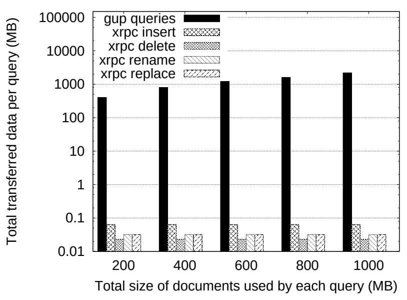


Fig. 14. Updates Bandwidth Usage

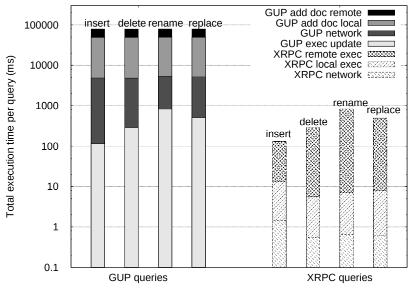


Fig. 15. Updates Time Breakdown (1000MB)

corresponds to the generic strategy discussed in Section VII where a remote document is first retrieved to the local machine (with fn:doc()), then the updates are applied on the local copy of the remote document, and finally the updated document is written to the remote peer using fn:put(). We call queries in this group "GUP *queries*" (i.e., Get-Update-Put). In the second group, called "XRPC *queries*", updates are applied directly on the original document at the remote peer with XRPC using so-called *updating functions* as specified by the XQUF:

```
module namespace fcn = "tkde";
declare updating function fcn:doInsert($doc as xs:string, $node as node())
{ do insert $node into doc($doc)/site };
declare updating function fcn:doDelete($doc as xs:string, $pid as xs:string)
{ do delete doc($doc)//person[./@id=$pid] };
declare updating function fcn:doRename($doc as xs:string, $pid as xs:string,
                                        $newname as xs:string)
{ do rename doc($doc)//person[@id=$pid] into $newname };
declare updating function fcn:doReplace($doc as xs:string, $pid as xs:string,
                                        $newnode as node())
{ do replace doc($doc)//person[@id=$pid] with $newnode };
```

We tested all four kinds of updates, keeping the granularity of the updates constant, affecting 100 person nodes. For example, the insert query in XRPC looks as follows:

```
import module namespace fcn="tkde" at "http://example.org/tkde.xq";
fcn:doInsert("xrpc://p2/xmark200mb.xml", doc("Persons100.xml")/persons)
```

All updating queries were applied on XMark documents of 200, 400, 600, 800 and 1000 MB, respectively. The data set is stored on one peer, which acts as the remote peer. The total execution time of all queries are shown in Figure 13.

For all four kinds of update queries, XRPC is significantly faster than GUP. The relatively small performance differences between different kinds of updates reflects the MonetDB/XQuery implementation of the XQUF. In Figure 15, the left-hand-side bars show the time breakdown of GUP queries, while the right-hand-side bars show the time breakdown of XRPC queries. We can conclude that with increasing document sizes, the absolute benefits of XRPC grow linearly, which is caused by the additional full serialisation, network copy, and shredding for the "Get" phase, followed by full serialisation and network copy steps in the "Put" phase, performed by the GUP approach. As the amount of updates is small, the total bandwidth usage of all GUP queries are approximately twice the documents size, as shown in Figure 14, whereas the XRPC query only sends the function parameters and results (tens of KB).

We finally recall that in all experiments (including the read-only ones) we used a local area network (LAN); but in a WAN environment, where much lower network performance is common, the benefits of our query decomposition techniques will be larger, as we showed by their strongly reduced network bandwidth use.

IX. Related Work

There are three main areas that are related to our proposal in this paper: distributed query processing, query decomposition and XML projection.

Much previous work in distributed query processing is surveyed in [19], [37] and parts of the book [25]. In distributed XML query processing, DXQ [13] depends on distributed query plans, in terms of the internal Galax execution algebra, generated by the Galax optimiser. In this respect, XRPC differs with its focus on interoperability, as it acts as a pure XQUERY rewriter (not making any assumptions on the system internals of the participating peers). Galax Yoo-Hoo [24] accesses web services using SOAP RPC as the communication protocol, which lacks proper support for XML elements and sequences; a problem addressed by XRPC using a specific literal SOAP message encoding. Active XML (AXML) [1] is a declarative framework that harnesses web services for data integration in a P2P architecture. Like XRPC, it also uses a (document/literal encoding) SOAP protocol to represent XML subtree values. However, the focus in AXML has been in adaptive call materialisation strategies, not on automatic query decomposition and the semantic challenges this brings in XQUERY, such as distributed node identity. XQueryD [28] supports function shipping in XQUERY like XRPC, but it does not define an open network protocol.

Decomposing queries to address multiple data sources is by now a well-studied problem in relational [34] and object-oriented [15], [20] DBMS. Many of these ideas and methods can be applied to XQUERY, yet we have shown here that issues of efficiently managing distributed node identity and document order add interesting challenges. [30], [31] discuss the decomposition of unstructured query languages only on a semi-structured database (a rooted, labeled graph). In XML databases, previous approaches require structural information about peers for supervising decomposition [33]. Other works, e.g., [6], [8], [32] only focus on a restricted subset of XQUERY queries. This paper is based on our previous work on decomposing read-only XQuery queries [40]. As one of the main goals of our work is to support *full-fledged* XQuery, in this paper, we have made our framework more complete by extending [40] with analysis on how updating queries can be decomposed. Some preliminary experiments have been done to show the performance improvements that could be achieved by executing updates directly on remote documents at the remote peers, instead of first retrieving a remote document to the local peer, then applying updates on the local copy of the document, and finally put the updated document back to the remote peer.

XML projection [21] drastically reduces the size of the data model representation using compile-time query characterisation. [5] introduces a precise XML pruning technique for a subset of XQUERY FLWOR expressions, based on the *apriori* knowledge of a data guide for underlying XML data. However, it does not handle XPATH predicates, backward axes and XQUERY-like languages. A type-based XML projection technique [2] is studied to improve current solutions with comparable or higher precision and less pruning overhead, as well as supporting backward XPATH axes. However, a DTD is required. [18] discusses runtime XML projection techniques. Based on the static compilation of runtime lookup-tables and a runtime-automaton from projection paths and a DTD, they can filter the input XML document efficiently using string matching algorithms. This technique, however, still lacks support for reverse XPATH axes and XQUERY built-in functions.

X. Conclusion

We have described a framework for distributed execution of full-fledged XQUERY including XQUF, focusing on the issue of providing equivalent query decompositions, in the face of semantic differences when (parts of) nodes are shipped across the network in XML messages. We first carefully characterised the problems that may occur regarding node identity and structural XPATH relationships in such a distributed setting. Then, we proposed a series of techniques such as pass-by-fragment and the use of a novel runtime XML projection method for serialising XML messages, that remove virtually all semantic problems and strongly improve performance, as shown by experiments on the open-source MonetDB/XQuery XDBMS (monetdb.cwi.nl). We also discussed the semantics of updating both local and remote documents using XQUF expressions, and additional constraints that should be added to the proposed techniques to guarantee semantic equivalence for such queries. Our main future work is an issue left out-of-scope here: deciding on distributed query placement after decomposition. In this area, we also contemplate using runtime methods to improve optimisation quality.

References

[1] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In *SIGMOD*, 2004.
[2] V. Benzaken et al. Type-Based XML Projection. In *VLDB*, 2006.
[3] S. Boag et al. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation 8 June 2006.
[4] P. Boncz et al. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, 2006.
[5] S. Bressan et al. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2), 2005.
[6] P. Buneman et al. Using Partial Evaluation in Distributed Query Evaluation. In *VLDB*, 2006.
[7] D. Chamberlin et al. XQuery Update Facility 1.0. W3C Candidate Recommendation 1 August 2008.
[8] G. Cong et al. Distributed query evaluation with performance guarantees. In *SIGMOD*, 2007.
[9] DataDirect XQuery. http://www.datadirect.com.
[10] D. Draper et al. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation 23 January 2007.
[11] eXist. http://exist.sourceforge.org.
[12] M. Fernández et al. XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Candidate Recommendation 11 July 2006.
[13] M. Fernández et al. Highly Distributed XQuery with DXQ. In *SIGMOD*, 2007.
[14] T. Grust et al. XQuery on SQL Hosts. In *VLDB*, 2004.
[15] V. Josifovski et al. Query decomposition for a distributed object-oriented mediator system. *Distributed and Parallel Databases*, 11(3), 2002.
[16] M. Kay. SAXON The XSLT and XQuery Processor. http://saxon.sourceforge.net.
[17] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. *SIGPLAN Not.*, 34(8):13–24, 1999.
[18] C. Koch et al. XML Prefiltering as a String Matching Problem. In *ICDE*, 2008.
[19] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4), 2000.
[20] H. Kozankiewicz, K. Stencel, and K. Subieta. Distributed query optimization in the stack-based approach. In *HPCC*, 2005.
[21] A. Marian et al. Projecting XML Documents. In *VLDB*, 2003.
[22] N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer (Second Edition). W3C Recommendation 27 April 2007.
[23] MonetDB/XQuery. http://monetdb.cwi.nl.
[24] N. Onose and J. Siméon. XQuery at Your Web Service. In *WWW*, 2004.
[25] M. T. Özsu and P. Valduriez. *Principles of distributed database systems (2nd ed.)*. Prentice-Hall, Inc., NJ, USA, 1999.
[26] IBM DB2 pureXML. http://www-01.ibm.com/software/data/db2/xml/.
[27] Qizx. http://www.qizx.com.
[28] C. Re et al. Distributed XQuery. In *IIWeb*, September 2004.
[29] A. Schmidt et al. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.
[30] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *VLDB*, 1996.
[31] D. Suciu. Distributed query evaluation on semistructured data. *ACM Trans. Database Syst.*, 27(1), 2002.
[32] K. Tajima and Y. Fukui. Answering XPath queries over networks by sending minimal views. In *VLDB*, 2004.
[33] L. T. T. Thuy, D. D. Duong, V. C. Bhavsar, and H. Boley. A bottom-up strategy for query decomposition. In *ICDIM*, 2006.
[34] E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Trans. Database Syst.*, 1(3):223–241, 1976.
[35] Web Services Atomic Transaction, August 2005. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.
[36] XQilla. http://xqilla.sourceforge.net/.
[37] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4), 1984.
[38] Y. Zhang and P. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *VLDB*, 2007.
[39] Y. Zhang and P. Boncz. Distributed XQuery and updates processing with heterogeneous XQuery engines. In *SIGMOD*, 2008.
[40] Y. Zhang, N. Tang, and P. Boncz. Efficient Distribution of Full-Fledged XQuery. In *ICDE*, 2009.